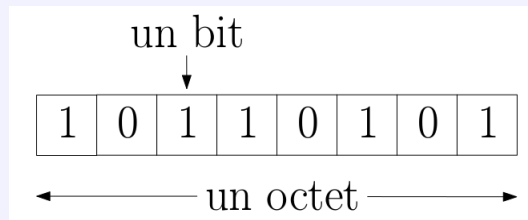


Introduction

La mémoire des ordinateurs est constituée d'une multitude de petits circuits électroniques (des transistors) et chacun ne peut être que dans deux états électriques : notés arbitrairement 0 et 1.

La valeur 0 ou 1 d'un circuit mémoire élémentaire s'appelle un **chiffre binaire**, un **booléen** ou un **bit** (abréviation de **binary digit**).

Depuis les années 1970, l'unité de mesure de l'espace (disque ou mémoire) est l'**octet** ou **byte**. Un octet correspond à 8 bits. Un octet peut donc prendre $2^8 = 256$ valeurs différentes.



En pratique, le processeur d'un ordinateur échange avec la mémoire des informations (données ou adresse mémoire) comportant plusieurs octets de 4 à 8 en général. On parle de **mots** qui caractérisent l'architecture de la machine (32 bits ou 64 bits).

Tout type d'information (nombre, caractère, couleur ...) peut être stocké sous forme de séquence de bits. Une **représentation interne** ou **codage** est nécessaire pour définir une représentation binaire. Dans ce chapitre, nous aborderons les codages simples des nombres entiers positifs et des entiers relatifs.

1 Représentation interne des entiers naturels

1.1 Représentation en base 10

Exemple 1

L'entier naturel 7307 s'écrit en base 10 comme une séquence de trois chiffres dont le poids dépend de leur position dans l'écriture.

Séquence de chiffres	7	3	0	7
Position	3	2	1	0
Poids	10^3	10^2	10^1	10^0

La décomposition de 7307 en base 10 s'écrit donc : $7307 = 7 \times 10^3 + 3 \times 10^2 + 0 \times 10^1 + 7 \times 10^0$.

Exercice 1

1. Écrire en Python une fonction `chiffres2nombre(t)` qui renvoie l'entier naturel correspondant à la liste de ses chiffres en base 10.

Les chiffres sont ordonnés dans un tableau par poids décroissant comme dans la représentation usuelle.

```
In [2]: chiffres2nombre([7,3,4])
Out [2]: 734
```

.....

.....

.....

.....

.....

2. On considère l’algorithme d’Horner décrit ci-dessous :

```
Fonction horner(liste)
    nombre prend la valeur 0
    Pour chiffre dans liste #on commence par les chiffres de poids forts
        nombre prend la valeur nombre x 10
        nombre prend la valeur nombre + chiffre
    Renvoyer nombre
```

a. Dérouler l’exécution de cet algorithme appliqué au tableau [7, 3, 0, 7].

.....

.....

.....

.....

b. Implémenter en Python la fonction horner. Que fait cette fonction ?

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

Exercice 2

On donne ci-dessous une suite d’instructions permettant de collecter dans un tableau les chiffres de 73 en base 10.

```
In [15]: (n, t) = (73, [])
```

```
In [16]: t.append(n % 10)

In [17]: n = n // 10

In [18]: (n, t)
Out[18]: (7, [3])

In [19]: t.append(n % 10)

In [20]: n = n // 10

In [21]: (n, t)
Out[21]: (0, [3, 7])

In [22]: t.reverse()

In [23]: t
Out[23]: [7, 3]
```

1. Écrire en pseudo-code une fonction qui prend en paramètre un entier et renvoie le tableau de ses chiffres en base 10.

.....

.....

.....

.....

.....

.....

.....

.....

2. Implémenter cette fonction en Python

.....

.....

.....

.....

.....

.....

.....

.....

1.2 Représentation en base 2



Théorème-Définition 1

La mémoire d'un ordinateur ne pouvant stocker que des séquences de bits à deux états, pour représenter un entier naturel en machine, il faut l'écrire en base deux où les seuls chiffres disponibles sont 0 et 1. Pour se convaincre que tout entier naturel peut s'écrire en base deux, on pourra visionner la vidéo sur cette page <http://video.math.cnrs.fr/magie-en-base-deux/>.

Tout entier naturel n peut s'écrire de façon unique en base 2 sous la forme $b_k b_{k-1} \cdots b_1 b_0$ telle que :

$$n = \sum_{i=0}^k b_i \times 2^i \text{ avec } \forall i \in \llbracket 0; k \rrbracket, b_i \in \llbracket 0; 1 \rrbracket$$

b_k est le bit de poids fort et b_0 est le bit de poids faible. Pour distinguer l'écriture en base 2 de l'écriture en base 10 on pourra écrire $\overline{101}^2$ pour par exemple l'écriture de 5 en base 2.



Exercice 3

1. Expliquer la formule suivante : « Il existe 10 catégories de personnes : celles qui comprennent le binaire et les autres.»

.....

2. Compter jusqu'à 18 en binaire.

.....
.....
.....

3. Quel est le plus grand entier non signé qu'on peut représenter en base 2 sur 64 bits? sur n bits?

.....

4. C'est en $\overline{11110010000}^2$ qu'Alan Turing a défini sa machine à calculer universelle. Et en base dix?

.....

5. Écrire en Python une fonction `bits2nombre(t)` qui renvoie l'entier naturel (en base dix) représenté par le tableau de bits `t` avec les bits de poids fort à gauche.

```
In [26]: bits2nombre([1,1,0,1])  
Out[26]: 13
```

.....

4. Compléter le code de la fonction `codage_binaire2(n)` qui renvoie un tableau de 0 ou de 1 représentant le codage binaire de l'entier `n` écrit en base 10 avec l'algorithme des *divisions successives*.

```
def codageBinaire2(n):  
    binaire = []  
    while n >= 2:  
        .....  
        .....  
        .....  
        .....  
        .....  
    binaire.append(n)  
    binaire.reverse()  
    return binaire
```

Exercice 5

1. a. $\overline{10110}^{(2)} + \overline{1101}^{(2)}$ | b. $\overline{100111}^{(2)} \times \overline{101}^{(2)}$

.....
.....
.....
.....
.....

2. Écrire en Python une fonction `additionBinaire8bits(t1, t2)` qui renvoie le tableau `t3` des bits de la somme des entiers représentés par les tableaux de bits `t1` et `t2`.

`t1`, `t2` et `t3` doivent être des tableaux de taille 8 correspondant à la représentation d'un entier sur un octet avec les bits de poids forts à gauche. Donner un exemple d'exécution où la fonction renvoie un résultat faux. Expliquer pourquoi.

```
In [29]: additionBinaire8bits([1,0,1,0,1,1,1,0], [0,0,0,0,1,1,1,1])  
Out[29]: [1, 0, 1, 1, 1, 1, 0, 1]
```

.....
.....
.....
.....
.....
.....

Méthode

- En Python, les entiers naturels et relatifs sont de type `int` et sont représentés de façon exacte, la seule limitation étant celle de la mémoire disponible.

```
In [2]: (type(734), type(-1))
Out[2]: (int, int)
```

- En Python, pour convertir l'écriture d'un entier naturel de la base 2 vers la base 10, on préfixe la séquence de bits de `0b` :

```
In [35]: 0b101
Out[35]: 5
```

- Pour convertir l'écriture d'un entier naturel de la base 10 vers la base 2, on utilise la fonction `bin` :

```
In [36]: bin(5)
Out[36]: '0b101'
```

1.3 Représentation en base 16 **Théorème-Définition 2**

Plus généralement, on peut représenter un entier naturel dans n'importe quelle base.

Une base très utilisée en informatique est la **base 16** ou **hexadécimale**.

On étend les dix chiffres de la base 10 avec A, B, C, D, E et F pour représenter 10, 11, 12, 13, 14 et 15.

En base 16 chaque octet est représenté par deux chiffres ce qui permet de condenser l'écriture.

Par exemple, chaque carte réseau possède une adresse MAC codée sur 48 bits soit 6 octets et représentée par une séquence de 6 chiffres en base 16 séparés par le caractère `:` sous la forme `:c8:60:00:a4:89:ab`

De même, les couleurs en représentation (R,G,B) sont codées sur 3 octets et dans le langage HTML des pages Web on les rencontre souvent notées comme séquence de six chiffres en base 16 préfixés par le caractère `#` : par exemple `#FF0000` va coder un rouge pur.

Exercice 6

- Convertir en base dix les entiers représentés en base 16 par $\overline{1A}^{16}$, \overline{AF}^{16} et \overline{FF}^{16} .

.....
.....

- La représentation hexadécimale s'obtient facilement à partir de la représentation binaire. Puisque $2^4 = 16$, pour convertir un octet de binaire en hexadécimal, il suffit de regrouper les bits par 4.

Ainsi 154 représenté en binaire sur un octet par $\underbrace{1001}_2^{\overline{9}^{16}} \underbrace{1010}_2^{\overline{A}^{16}}$ a pour représentation hexadécimale $\overline{9A}^{16}$ car $9 \times 16 + 10 = 154$.

Convertir de même en hexadécimal les octets suivants :



$$\begin{array}{r}
 \boxed{0} \boxed{0} \boxed{1} \boxed{1} \boxed{0} \boxed{1} \boxed{1} \boxed{1} \rightarrow 55 \\
 + \boxed{1} \boxed{0} \boxed{0} \boxed{1} \boxed{0} \boxed{1} \boxed{0} \boxed{1} \rightarrow -21 \\
 = \boxed{} \boxed{} \boxed{} \boxed{} \boxed{} \boxed{} \boxed{} \boxed{} \rightarrow
 \end{array}$$

À quel problème est-on confronté?

.....

.....

.....

.....

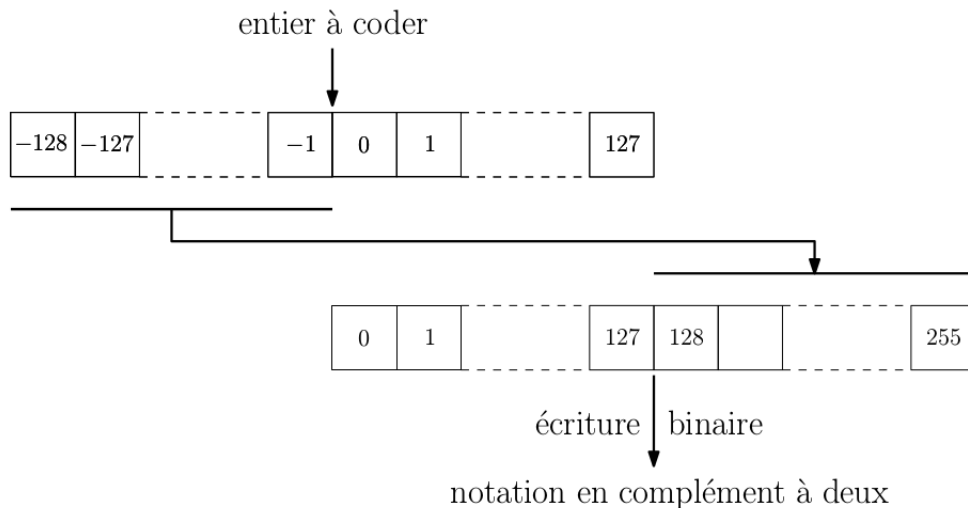
2.2 Par complément à 2

Méthode Complément à 2

Commençons par regarder le cas particulier des entiers codés sur un octet.

La **représentation en complément à 2** consiste à représenter les entiers entre -128 et 127 (au lieu de 0 à 255) de la manière suivante :

- si $x \in [0; 127]$ alors x est codé par son écriture binaire sur un octet.
- si $x \in [-128; -1]$ alors x est codé par l'écriture binaire de $x + 256$ sur un octet.



Ainsi, on fait correspondre à chaque entier de $[-128; 127]$ un entier de $[0; 255]$ qui est ensuite codé sur 8 bits par sa représentation binaire. On remarquera que 0 admet alors une seule écriture : $\overline{00000000}^2$. Par exemple la représentation en complément à deux sur 8 bits de -43 est $\overline{11010101}^2$ et celle de 117 est $\overline{01110101}^2$.


.....

.....

.....

.....

.....

 ***Ainsi il n'est pas nécessaire d'avoir deux algorithmes distincts pour l'addition et la soustraction. Par rapport à la représentation par signe et valeur absolue, cela se traduit dans l'implémentation matérielle de l'addition par une réduction du nombre de portes logiques et donc une économie de transistor et une simplification des circuits.***

3. Que donne l'addition des représentations de 7 et 1 sur 4 bits en complément à deux? Commenter

.....

.....

.....

.....

.....

4. Pour obtenir la représentation en complément à 2 sur n bits de l'opposé d'un entier naturel inférieur à $2^{n-1} - 1$, il suffit de prendre le complément de chaque bit de sa représentation binaire ($0 \rightarrow 1$ et $1 \rightarrow 0$) et d'ajouter 1. Justifier cet algorithme à partir de la définition de la représentation en complément à 2.

.....

.....

.....

.....

.....

5. La classe `int8` du module `numpy` permet de créer des entiers signés codés sur 8 bits. Commenter les résultat des exécutions ci-dessous :

```
In [12]: import numpy

In [13]: numpy.int8(127) + numpy.int8(1)
__main__:1: RuntimeWarning: overflow encountered in byte_scalars
Out[13]: -128
```

```
In [14]: numpy.int8(-127) + numpy.int8(-2)
__main__:1: RuntimeWarning: overflow encountered in byte_scalars
Out[14]: 127
```

6. Écrire un jeu de tests unitaire pour la fonction `complement_deux(n, nbits)` dont on donne la spécification ci-dessous puis coder la fonction. On utilisera la fonction `codage_binaire2(n)` écrite à l'exercice 4.

```
from typing import List

def complement_deux(n:int, nbits:int)->List[int]:
    """
    Renvoie la notation en compléments à 2 de l'entier signé n
    sous la forme d'un tableau de bits ordonnés de gauche à droite
    par poids décroissant

    Parameters
    -----
    n : int    Précondition  $-2^{**}(nbits-1) \leq n < 2^{**}(nbits-1)$ 
    nbits : int

    Returns
    -----
    List[int] tableau de nbits bits
    """
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....

# Jeu de tests unitaires
assert complement_deux(0, 8) == [0, 0, 0, 0, 0, 0, 0, 0]
assert complement_deux(5, 8) == [0, 0, 0, 0, 0, 1, 0, 1]
assert complement_deux(2**7 - 1, 8) == [0, 1, 1, 1, 1, 1, 1, 1]
assert complement_deux(-2**7, 8) == [1, 0, 0, 0, 0, 0, 0, 0]
assert complement_deux(2**7 - 2, 8) == [0, 1, 1, 1, 1, 1, 1, 0]
assert complement_deux(-2**7 + 1, 8) == [1, 0, 0, 0, 0, 0, 0, 1]
assert complement_deux(-1, 8) == [1, 1, 1, 1, 1, 1, 1, 1]
assert complement_deux(-2, 8) == [1, 1, 1, 1, 1, 1, 1, 0]
```