Chapitre 6 : Représentation des nombres

ISN

Année scolaire 2019/2020

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, un ordinateur manipule des **mots machines** de plusieurs octets. La taille mot correspond au nombre d'adressages possibles de la mémoire vive RAM. On rappelle que dans l'architecture de Von Neumann, la mémoire contient des données et des programmes et en particulier un **mot machine** peut contenir l'adresse d'un autre mot mémoire ...
Les **mots machines** sont de nos jours le plus souvent de 64 bits.
Tout type d'information (nombre, caractère, couleur ...) peut être stocké sous forme de séquence de bits. Un **encodage** est nécessaire pour définir une représentation binaire. Dans ce chapitre, nous aborderons les encodages simples des nombres entiers, entiers relatifs ou des réels.

I Encodage des entiers naturels

I.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 `chiffes2nombre(t)` qui retourne l'entier naturel correspondant à la liste de ses chiffres en base 10.

Les chiffres sont ordonnés dans la liste 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
    Retourner nombre
```

a. Dérouler l'exécution de cet algorithme appliqué à la liste `[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 une liste 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 argument un entier et retourne la liste de ses chiffres en base 10.

.....

.....

.....

.....

.....

.....

2. Implémenter cette fonction en `Python`

.....

.....

.....

.....

.....

.....

I.2 Représentation en base 2

Théorème-Définition 1

La mémoire d'un ordinateur ne pouvant stocker que des séquences de bit pouvant prendre uniquement 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 :

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

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 avec un mot de 64 bits ? de n bits ?

   .....

   .....

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

   .....

   .....

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

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

.....

.....

.....

.....

.....

.....

Exercice 4

1. Regarder la vidéo sur la page http://video.math.cnrs.fr/magie-en-base-deux/.

2. En utilisant les deux algorithmes présentés dans la video, convertir en base 2 les entiers naturels d'écritures décimales 37 et 18.

.....

.....

.....

.....

.....

3. Compléter le code de la fonction `codageBinaireGlouton(n)` qui retourne une liste de 0 ou de 1 représentant le codage binaire de l'entier n écrit en base 10 en appliquant l'algorithme glouton décrit dans la vidéo.

```
def codageBinaireGlouton(n):
    binaire = []
    puissance2 = 1
    while puissance2 <= n:
        puissance2 = puissance2 * 2
    ..............................
    ..............................
    ..............................
    ..............................
    ..............................
    ..............................
    return binaire
```

4. Compléter le code de la fonction `codageBinaire2(n)` qui retourne une liste de 0 ou de 1 représentant le codage binaire de l'entier n écrit en base 10 en appliquant l'algorithme des divisions

successives décrit dans la vidéo.

```
def codageBinaire2(n):
    binaire = []
    while n >= 2:
        ..................................
        ..................................
        ..................................
        ..................................
        ..................................
    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 retourne la liste `t3` des bits de la somme des entiers représentés par les listes de bits `t1` et `t2`.

`t1`, `t2` et `t3` doivent être des listes 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 retourne 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'
```

I.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.
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

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 $\overbrace{1001}^{\overline{9}^{16}}{}^{2}\overbrace{1010}^{\overline{A}^{16}}{}^{2}$ a pour représentation hexadécimale $\overline{9A}^{16}$ car $9 \times 16 + 10 = 154$.
Convertir de même en hexadécimal les octets suivants :

- 11101010
- 10011011
- 1100000

.....

.....

.....

.....

Méthode

☞ En `Python`, pour convertir l'écriture d'un entier naturel de la base 16 vers la base 10, on préfixe la séquence de bits de `0x` :

```
In [5]: 0x9A
Out[5]: 154
```

☞ Pour convertir l'écriture d'un entier naturel de la base 10 vers la base 16, on utilise la fonction `hex` :

```
In [6]: hex(154)
Out[6]: '0x9a'
```

II Encodage des entiers relatifs

Exercice 7

On souhaite représenter des entiers relatifs sur trois bits.

1. Combien d'entiers relatifs peut-on représenter sur trois bits ?

.....

2. On décide de représenter le signe d'un entier par le bit de poids fort (0 pour positif et 1 pour négatif) et de garder les deux autres bits pour représenter la valeur absolue du nombre.

a. Quels entiers peut-on représenter sur 3 bits avec cet encodage ?

.....

.....

.....

.....

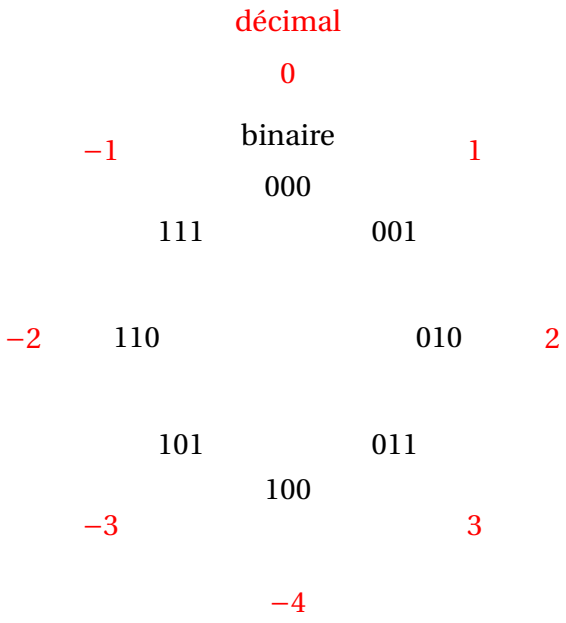
b. Additionner les représentations de 1 et de -1 dans cet encodage ? Que remarque-t-on ?

.....

.....

.....

3. L'encodage par complément à 2 des entiers relatifs consiste à garder le bit de poids fort pour le signe, à coder les positifs comme précédemment et à coder les négatifs en prenant pour les deux bits de poids faible leur inverse (0 pour 1 et 1 pour 0) et à ajouter 1 au nombre obtenu sans tenir compte de la retenue. La figure ci-dessous illustre l'encodage en complément à 2 sur 3 bits.



a. Vérifier si l'addition des représentations de 1 et de −1 dans cet encodage donne bien 0 et si l'addition des représentations de −1 et de −1 dans cet encodage donne bien −2 (en excluant la dernière retenue sortante).

.....

.....

.....

.....

b. Que donne l'addition des représentations de 3 et 1 sur 3 bits ? Commenter

.....

.....

c. Déterminer les entiers relatifs qu'on peut représenter en complément à 2 sur n bits ?

.....

.....

.....

d. Quels entiers relatifs peut-on représenter en complément à 2 sur n bits ?

.....

.....

.....

e. Comment peut-on obtenir l'encodage en complément à 2 d'un entier naturel inférieur à $2^n - 1$ sur n bits ?

.....

.....

.....

f. 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
```

.....

.....

.....

III Encodage des réels

Exercice 8

1. Quel est le type de représentation des nombres réels en `Python`?

```
In [15]: (type(1),type(1.0), type(4 // 2), type(4/2))
Out[15]: (int, float, int, float)
```

.....

.....

.....

2. Commenter les résultats d'exécution ci-dessous :

```
In [1]: 0. + 0.2 == 0.3
Out[1]: False

In [2]: 1.6 + (3.2 + 1.7)
Out[2]: 6.5

In [3]: (1.6 + 3.2) + 1.7
Out[3]: 6.500000000000001
```

```
In [4]: 1.5 *(3.2 + 1.4)
Out[4]: 6.8999999999999995

In [5]: 1.5 * 3.2 + 1.5 * 1.4
Out[5]: 6.9

In [6]: x = 1e200

In [7]: x * x
Out[7]: inf

In [8]: (x * x) * 0
Out[8]: nan

In [9]: y = 10 ** 400

In [10]: type(y + 1.0)
OverflowError                             Traceback (most recent call last)
<ipython-input-49-01ee37eb4b77> in <module>
----> 1 type(y + 1.0)
OverflowError: int too large to convert to float

In [10]: type(y + 1)
Out[10]: int
```

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

3. Lire l'article disponible sur la page Web http://www.apprendre-en-ligne.net/bloginfo/index.php/2008/10/10/79-problemes-d-arrondi. Quels problèmes peuvent causer la représentation approchée des nombres réels en informatique ?

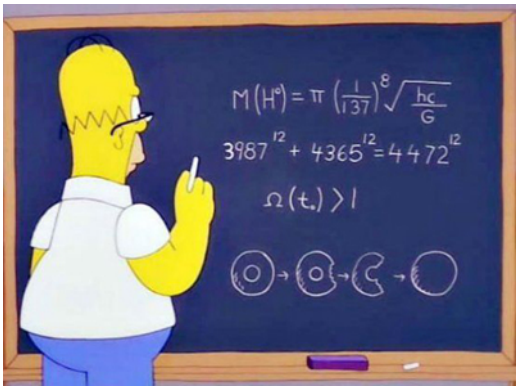
.....

.....

.....

.....

4. Dans l'épisode 2 de la saison 10 des Simpson (1998), intitulé "La Dernière Invention d'Homer", on peut voir le tableau ci-contre. Rappelons que le grand théorème de Fermat, démontré par Andrew Wiles, affirme que pour tout entier $n > 2$, il n'existe pas de nombres entiers strictement positifs x, y et z tels que $x^n + y^n = z^n$. Par défaut en `Python`, les flottants sont représentés en double précision sur 64 bits et les entiers sont en précision infinie. Le module `numpy` propose un type de flottants sur 32 bits.



Évaluez les tests suivants et commentez.

```
In [34]: import numpy as np

In [35]: np.float32(np.float32(3987) ** 12 + np.float32(4365) ** 12)
      == np.float32(np.float32(4472) ** 12)
Out[35]: ........

In [36]: float(3987) ** 12 + float(4365) ** 12 == float(4472) ** 12
Out[36]: .....

In [37]: 3987 ** 12 + 4365 ** 12 == 4472 ** 12
Out[37]: ........
```

.....

.....

.....

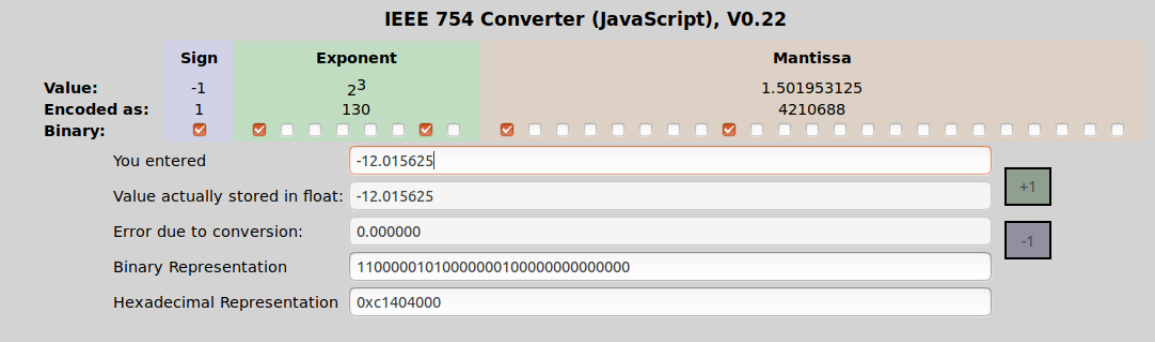
Théorème-Définition 3

Les nombres réels sont représentés usuellement en informatique sous la forme de **flottants** en référence au mode de représentation en virgule flottante similaire à la notation scientifique en base dix où un réel x est représenté sous la forme d'un triplet (signe s, mantisse m, exposant n) avec $x = (-1)^s \times m \times 10^n$.
La représentation des flottants est fixée par la norme IEEE 754 : les différences avec la notation scientifique sont la base (2 pour les flottants) et l'exposant n qui est décalé avec un biais d qui permet de représenter des exposants négatifs et positifs comme des entiers.

Un flottant f est alors de la forme $\boxed{f = (-1)^s \times m \times 2^{n-d}}$ avec $m \in [1\,;2[$.

Par exemple pour $-12.015625 = -2^3 \times \left(1 + \frac{1}{2} + \frac{1}{2^9}\right)$ le signe est -1, l'exposant est 3 et la mantisse est $\frac{1}{2} + \frac{1}{2^9}$.
On donne ci-après la représentation de -12.015625 comme flottant au format simple précision sur 32 bits obtenue avec le simulateur https://www.h-schmidt.net/FloatConverter/IEEE754.html.
En précision 32 bits, le signe s se code sur le bit de poids fort, puis l'exposant est codé sur 8 bits avec un biais $d = 2^{8-1} - 1 = 127$ et une mantisse codée sur 23 bits.
Par défaut le premier bit de la mantisse n'est pas codé et considéré comme égal à 1.
Comment peut-on représenter le 0 ? La convention est de réserver les valeurs extrêmes de l'exposant non

biaisé e (0 et 255 pour un codage sur 8 bits) aux valeurs particulières comme 0 (exposant 0) ou l'infini (exposant 255).



Les flottants permettent de représenter des nombres très grands ou très petits. `Python` est particulier car les entiers peuvent être arbitrairement grands mais le coût en espace mémoire est important. Mais les flottants ne sont que des représentations approchées des réels et il existe un écart non nul entre deux flottants consécutifs, un écart élastique qui dépasse 1 au delà de 2^{52}. Par conséquent il faut éviter de tester l'égalité entre deux flottants a et b mais plutôt tester une inégalité $|a-b|<\varepsilon$.

Exercice 9

En 1202, *Leonardo Fibonacci* a donné sans démonstration, dans son ouvrage *Liber Abaci* un algorithme permettant d'écrire toute fraction d'entiers positive $\frac{p}{q}$ avec $\frac{p}{q}\in]0\,;1[$ sous forme d'une somme de fractions égyptiennes, de numérateur 1 :

> « *Soustraire à la fraction donnée la plus grande fraction égyptienne possible, répéter l'opération avec la nouvelle fraction, et ainsi de suite jusqu'à obtenir une fraction égyptienne.* »

Il s'agit d'un algorithme glouton car à chaque étape, il réalise un choix localement optimal : la plus grande fraction égyptienne inférieure ou égale à la fraction restante.

Par exemple on a $\frac{2}{3}=\frac{1}{2}+\frac{1}{6}$ ou $\frac{2}{59}=\frac{1}{36}+\frac{1}{236}+\frac{1}{531}$.

1. Appliquer cet algorithme à la main pour obtenir une décomposition de $\frac{2}{5}$ puis de $\frac{51}{105}$.
2. On considère les deux fonctions de décomposition ci-après. Appeler ces fonctions avec les arguments `3/4` puis `2/5`. Que peut-on observer ? Comment l'expliquer ?
3. On choisit désormais de représenter une fraction non pas sous forme d'approximation décimale (flottant en `Python`) mais sous la forme du couple d'entiers constitué par leur numérateur et leur dénominateur. Par exemple $\frac{4}{17}=\frac{p}{q}$ sera représenté par le couple d'entiers $p=4$ et $q=17$ et non par le flottant `0.23529411764705882`.
4. **a.** Soit p et q le numérateur et le dénominateur respectifs de la fraction au début d'une itération fixée de la boucle externe et soit n le dénominateur correspondant au choix glouton. Exprimer le numérateur et le dénominateur de la fraction restante en fonction de p, q et n. Tester les formules obtenues pour retrouver que $\frac{4}{17}=\frac{1}{5}+\frac{1}{29}+\frac{1}{1233}+\frac{1}{3039345}$.

   b. Écrire une fonction `decomposition3(p, q)` qui implémente l'algorithme avec cette représentation des fractions.

c. Écrire un programme qui détermine le plus grand dénominateur et le n pour lequel il est atteint sur l'ensemble des décompositions calculées par l'algorithme pour les fractions $\frac{4}{n}$ avec $5 \leqslant n \leqslant 288$.

d. Tester l'appel `decomposition3(4, 289)`. En cas d'erreur déboguer puis corriger le code. Indice : les calculs ne doivent manipuler que des entiers.

```
def decomposition1(x):
    decomp = []
    while x != 0:
        n = 1
        while 1/n > x:
            n += 1
        decomp.append(n)
        x = x - 1 / n
        print(x, decomp)
    return decomp
```

```
import math

def decomposition2(x):
    decomp = []
    while x != 0:
        n = math.ceil(1/x)
        decomp.append(n)
        x = x - 1 / n
        print(x, decomp)
    return decomp
```

Méthode

Pour la documentation Python sur les flottants lire l'article :
https://docs.python.org/fr/3.7/tutorial/floatingpoint.html.

- Le module `sys` permet d'obtenir des informations sur la précision de la représentation des flottants en Python (double précision sur 64 bits) :

```
In [52]: sys.float_info
Out[52]: sys.float_info(max=1.7976931348623157e+308, max_exp=1024,
     max_10_exp=308, min=2.2250738585072014e-308, min_exp=-1021,
     min_10_exp=-307, dig=15, mant_dig=53, epsilon=2.220446049250313
     e-16, radix=2, rounds=1)
```

On peut lire que les flottants représentables sont compris entre 10^{-308} et 10^{308} environ et que le plus petit écart entre deux flottants au voisinage de 1 est $2^{-52} \approx 2.220446049250313e-16$.

- Le module `decimal` permet d'obtenir la représentation approchée d'un réel sous forme de flottant.

```
In [53]: import decimal

In [54]: decimal.Decimal.from_float(0.1)
Out[54]: Decimal('
     0.1000000000000000055511151231257827021181583404541015625')
```