1 Représentation des nombres

Le « vol 501 », vol inaugural d'Ariane 5, eut lieu le 4 juin 1996. Il s'est soldé par l'explosion de la fusée 36 secondes après le décollage. Cette explosion est due à un bug informatique, plus exactement à une erreur liée à la représentation de nombres.

L'objectif de ce chapitre est de mieux comprendre ce qu'est un nombre pour un ordinateur et pour le langage Python, afin, entre autres, d'effectuer des choix éclairés lors de l'implémentation (codage dans un langage) d'algorithmes de calcul numérique. Pour comprendre les enjeux, on abordera les questions suivantes :

- qu'est-ce qu'un système de numération ?
- comment sont représentés les nombres entiers, naturels et relatifs ?
- comment sont effectuées les opérations entre entiers ?
- comment sont représentés les autres nombres ?
- quelles sont les conséquences des choix de ces représentations ?

1.1 Bases et systèmes de numération

1.1.1 Introduction

Un système de numération est l'ensemble des règles d'utilisation des nombres, ou plus exactement des symboles qui permettent de représenter des nombres. Le nombre quatorze était par exemple représenté par « XIV » chez les romains (numération romaine), alors qu'il est représenté par « 14 » dans notre **système de numération indo-arabe**. Les conséquences qu'a eues sur les société humaines la démocratisation de ce dernier système de numération sont incalculables et ont historiquement facilité les progrès scientifique. Ce système de numération, aussi dénommé système **décimal**, n'est qu'un exemple parmi d'autres de système de numération positionnelle dans une base donnée, ici **la base 10**.

1.1.2 Écriture d'un entier positif en base 10

Notons N l'entier huit mille sept cent soixante huit. Alors en **base 10** on peut décomposer N de la façon suivante :

$$\mathrm{N} = 8\,768 = 8 \times 10^3 + 7 \times 10^2 + 6 \times 10^1 + 8 \times 10^0$$

et de façon générale, un entier N s'écrit en base 10 sous la forme

$$c_n c_{n-1} \cdots c_2 c_1 c_0$$

où les c_i sont les chiffres de N, $(c_0, c_1, \ldots, c_n) \in [\![0;9]\!]^{n+1}$ et

$$N = c_n \times 10^n + c_{n-1} \times 10^{n-1} + \cdots + c_1 \times 10^1 + c_0 \times 10^0$$

et $n+1$ est le nombre de chiffres de N.
Au rang i correspond le chiffre (ou digit) c_i de poids 10^i en base 10 et un nombre de n chiffres est aussi appelé **mot** de taille n : $c_3c_2c_1c_0$ est un **mot** de **taille** quatre.
Cette écriture est dite positionnelle car, contrairement à la numération romaine où le chiffre I, qu'il soit avant ou après un V, a toujours la valeur un, la valeur d'un chiffre en notation décimale est liée à sa position dans le nombre.

1.1.3 Écriture d'un entier dans une base b

L'objectif est maintenant de comprendre qu'on peut utiliser une autre base que 10 (10 a été choisi uniquement parce que c'est le nombre de doigts d'un humain).

Définition – Écriture d'un entier dans une base b

Soit b un entier, $b \geqslant 2$. **L'écriture en base b** des entiers est une écriture positionnelle utilisant **b chiffres** (ou **digits**). Un nombre N a une écriture en base b notée N_b et celle-ci est de la forme

$$N_b = c_n c_{n-1} \cdots c_2 c_1 c_0$$

où, pour tout $i \in [\![0;n]\!]$, $c_i \in \{0, 1, \ldots, b-1\}$ et N a pour forme développée

$$N = c_n b^n + c_{n-1} b^{n-1} + \cdots + c_1 b^1 + c_0 b^0$$

Autre notation en base b :

$$N_b = [c_n; c_{n-1}; \ldots; c_1; c_0]_b$$

Remarque : si b=10 alors la base b n'est pas toujours précisée.

Exemple – Écriture d'un nombre en base 8

En base 8 les chiffres (digits) sont 0, 1, 2, 3, 4, 5, 6 et 7.
Ainsi $456_8 = 4 \times 8^2 + 5 \times 8 + 4 = 302$.
Cette base a été utilisée par les amérindiens Yukis, ils comptaient non pas avec leurs doigts mais avec l'espace entre les doigts.

Exemple – Écriture d'un nombre en base 16

En base 16, ou base **hexadécimale**, les digits sont 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E et F. Ainsi :

$$A3B_{16} = (10 \times 16^2 + 3 \times 16^1 + 11 \times 16^0)_{10} = 2619.$$

Par exemple la clé de sécurité des box Internet est écrite en hexadécimal.

Exemple – Écriture d'un nombre en base 60

Cette base, utilisée par les Sumériens, les Akkadiens puis les Babyloniens, est encore celle qui régit les heures, minutes et secondes ainsi que les angles en degrés. Par exemple $[1;25;12]_{60} = (1 \times 60^2 + 25 \times 60 + 12)_{10} = 5112$.

L'exemple fondamental du Binaire

Le système **binaire** est l'autre nom de la **base 2**, ce système d'écriture des nombres comporte donc uniquement deux symboles, **0 et 1**.

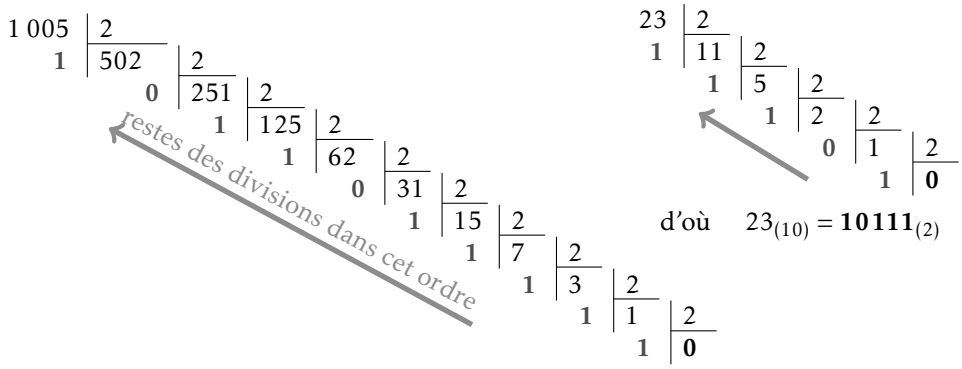
On utilise ce système en informatique car il n'y a besoin que de deux valeurs qui correspondent, en électronique, à deux niveaux de tension différents. D'autres systèmes auraient pu exister (système ternaire, décimal) mais les complications électroniques rencontrées sont trop nombreuses pour qu'ils soient mis en place à l'heure actuelle.

Voici des entiers écrits en base 10, en binaire et en base 16 :

Le même nombre écrit ...		
... en base 10	... en base 2	... en base 16
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
⋮	⋮	⋮
9	1001	9
10	1010	A
11	1011	B
12	1100	C
⋮	⋮	⋮
165	10100101	A5
⋮	⋮	⋮
255	11111111	FF

Complément – du décimal au binaire

Pour passer de l'écriture décimale à l'écriture binaire d'un entier N on peut effectuer des **divisions euclidiennes successives**, division de N par 2 puis des quotients successifs **par 2**, jusqu'à ce que le quotient devienne 0. L'écriture binaire de N est alors la suite des restes dans **l'ordre inverse** de leur obtention.



d'où $1\,005_{(10)} = \mathbf{1\,111\,101\,101}_{(2)}$

1.2 Représentation des nombres dans un ordinateur

1.2.1 Des digits et des mots pour représenter des nombres

Comme toute information dans un ordinateur, les nombres sont **codés à l'aide de bits** (***bi**nary digi**t***). Ces bits sont, sauf exception, regroupés par huit et un groupe de huit bits est appelé un **octet** (***byte*** en anglais, de symbole **B**, à ne pas confondre avec **b** pour bits car 1B = 8b).

Définition – mot et taille d'un mot

On appelle **mot** un groupe de bits (un n-uplet pour être plus exact).
Les digits sont aux nombres ce que les lettres sont aux mots. Par exemple un octet est un mot de **taille** 8. Un **mot de taille n** sera noté

$$b_{n-1}\,b_{n-2}\cdots b_1\,b_0$$

Exemple : 0100 1011 est un mot de taille 8 en base 2 représentant 75 en base 10.

C'est le type d'un nombre qui détermine sa représentation : les « entiers courts » du langage C sont de taille 16. En Python, les **entiers naturels** et les **entiers relatifs** sont normalement codés sur des mots de taille 64. Mais il existe aussi des « **entiers longs** » de taille . . . uniquement limitée par la mémoire de l'ordinateur.

1.2.2 Quatre types de représentation des nombres

Dans un ordinateur, les entiers naturels (0, 1, 2, . . .) et les entiers relatifs (. . . , -2, -1, 0, 1, 2, . . .) ont une représentation en binaire. Les entiers naturels sont aussi appelés entiers **non signés** (*unsigned*) et les relatifs sont les entiers signés. Bien que cela soit transparent pour l'utilisateur, il existe trois types de codage pour ces entiers :

- deux codages sur des mots de taille fixe :
  - un pour les entiers naturels,
  - un pour les entiers relatifs ;
- un dernier pour les entiers (relatifs) longs sur des mots de taille illimitée.

Les trois sections suivantes décrivent ces trois types de codage.
Une dernière représentation, très différente des autres et qui sera vue dans la toute dernière partie, est la représentation **à virgule flottante** ; elle permet de représenter des réels sur des mots de taille fixe (64 bits en général pour un PC actuel).

Un microprocesseur sait gérer ces différentes représentations, il comporte différentes zones de calcul, chacune dédiée aux opérations entre nombres d'un même type (de représentation). Les sections suivantes détaillent le codage de chacune de ces représentations.

1.3 Entiers naturels codés sur des mots de taille fixe

1.3.1 Le codage « naturel » des entiers positifs

Bien qu'en réalité la taille des mots soit souvent de 32 ou 64 bits, pour simplifier et parce que les principes en jeux sont indépendants de la taille des mots, les explications seront illustrées avec des mots de taille 8. Avec de tels mots on peut représenter 2^8 soit 256 entiers, donc les entiers de 0 à 255, soit en binaire de $0000\ 0000_2$ à $1111\ 1111_2$. Par exemple, le nombre $240_{10} = 1111\ 0000_2$ est représenté par

1	1	1	1	0	0	0	0

sur huit bits, et sur un mot de taille 16 il le sera par

0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0

Remarquons que $1111\ 1111_2 = \sum_{k=0}^{7} 2^k = \frac{1-2^8}{1-2} = 2^8 - 1 = 255$. Sur des mots de taille 32, on peut représenter $2^{32} = 4\,294\,967\,296$ entiers, le plus grand étant $2^{32}-1$ et avec des mots de taille 64 on peut représenter $2^{64} = 18446744073709551616$ entiers.

1.3.2 Opérations sur les entiers codés sur des mots de taille fixe

Dans un ordinateur, c'est l'unité arithmétique et logique du CPU (microprocesseur) qui effectue les opérations, une partie de cette zone du CPU étant dédiée aux calculs sur les entiers.

Addition et complexité

La notion de complexité sera étudiée en détail au chapitre 8 page 115 ; après une première lecture rapide, une relecture de ce paragraphe après lecture du chapitre 8 sera profitable.
Voici la table de vérité de l'addition en binaire :

Addition en base 2			
b_1	b_2	$b_1 + b_2$	Retenue
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Appliquons ceci sur des mots de taille 8 : l'addition 179 + 34 s'écrit alors en binaire

```
    1       1   <-- retenues
  1 0 1 1 0 0 1 1
+ 0 0 1 0 0 0 1 0
  1 1 0 1 0 1 0 1
```

Fig. 1.1 – L'addition 179 + 34 écrite et posée en base 2.

Il faut donc pour cette opération tenir compte des retenues. Si les deux mots sont de taille fixe n alors il y a n opérations élémentaires comme celles décrites par la table et *dans le pire des cas*, $n-1$ retenues, donc $2n-1$ additions.
Ainsi **la complexité de l'addition est en $\Theta(n)$, où n est la taille des nombres : elle est linéaire**.
Remarques :

- si la *dernière* addition engendre une retenue, cela signifie que la somme des deux mots de taille n est strictement supérieure à $2^n - 1$, donc n'est pas représentable par un mot de taille n. On parle alors d'*overflow* ou de **dépassement de capacité** ;
- la soustraction a elle-aussi une complexité en $\Theta(n)$ où n est la taille des mots.

Multiplication et complexité

De même, on pourrait écrire la table de la multiplication en binaire mais puisqu'il s'agit de ne multiplier que par des 1 et des 0, posons une telle multiplication comme à l'école élémentaire :

```
            1 1 0 1
×           1 0 1 1
          1 1 1 0 1      <-- retenue
        1 1 1 0 1 ·
      1 0 0 0 0 · ·
    1 1 0 1 · · ·
  1 0 0 0 1 1 1 1
```

Fig. 1.2 – Une addition en base 2.

Les résultats des 4 multiplications intermédiaires ne sont en fait que des copies du premier facteur 1101 décalées de 0 à 3 bits au plus vers la gauche. Il y a ensuite 3 sommes intermédiaires à calculer (on copie, on additionne, on copie, ...).
En généralisant, (*dans le pire des cas*) il faut créer n copies du premier facteur puis effectuer $n-1$ additions chacune ayant une complexité en $\Theta(n)$ donc la **multiplication de deux nombres de taille n a une complexité en $\Theta(n^2)$** : elle est quadratique.
Remarques :

- il existe d'autres façons simples de multiplier (multiplication antique égyptienne, russe, ...) mais leur complexité est elle aussi quadratique ;
- il existe, *pour les grand nombres uniquement*, des algorithmes permettant d'améliorer cette complexité. L'un de ceux-ci sera détaillé dans la section suivante. Il est notable que les algorithmes de meilleure complexité sont en $\Theta(n\log(n))$; ils ont été décrits en 2019 mais ne sont utilisables que pour des nombre supérieurs à $2^{1729^{12}}$: ils ne sont pas applicables en pratique !

1.4 Entiers relatifs codés sur des mots de taille fixe

Pour représenter les entiers relatifs, il faut étendre la représentation des entiers positifs aux négatifs ; cela est effectué au sein des PC par une méthode nommée **complément à deux**. La voici expliquée sur des mots de taille 8.

Sur de tels mots, il y a 2^8, soit 256 entiers relatifs distincts qui peuvent être représentés. La **moitié** de ces 256 mots est alors utilisée pour représenter les **entiers positifs**, l'**autre moitié** pour les **négatifs**. Il y aura donc 128 entiers positifs représentables, les nombres de 0 à 127 et 128 entiers négatifs représentables, les nombres de -128 à -1. Alors :

- les 128 **entiers positifs** (de 0 à $127 = 2^7 - 1$) sont représentés **par le même entier** ;
- les 128 **entiers x négatifs** ($-128 = -2^7 \leqslant x \leqslant -1$) sont représentés par l'entier positif $2^8 + x = 256 + x$, écrit en binaire (255 en binaire représente -1, ...).

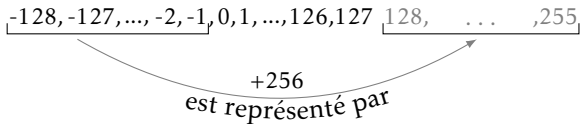


Fig. 1.3 – Sur des mots de taille 8 ($m = 8$), un entier négatif x est représenté par $x + 2^m = x + 256$; par exemple, -128 est représenté par $-128 + 256 = 128 = 1000\,000_2$, ... -1 est représenté par $-1 + 256 = 255 = 1111\,1111_2$.

Remarquons que, comme on a alors l'encadrement $128 \leqslant 256 + x \leqslant 255$, les entiers de 128 à 255 représenteront les 128 entiers négatifs x de -128 à -1.
Le tableau 1.1 donne la représentation d'entiers relatifs de -128 à $+127$ sur huit bits avec la méthode du complément à deux. Par exemple 1111 1110 représente -2 et 1000 0000 représente -128.

Entier relatif x	Entier non signé	**Représentation sur un mot de taille 8 x**
-128	128	**1000 0000**
-127	129	**1000 0001**
⋮	⋮	⋮
-3	253	**1111 1101**
-2	254	**1111 1110**
-1	255	**1111 1111**
0	0	**0000 0000**
1	1	**0000 0001**
⋮	⋮	⋮
126	126	**0111 1110**
127	127	**0111 1111**

Table 1.1 – Représentation sur 8 bits de 256 entiers relatifs par la méthode du complément à deux.

L'encadré qui suit donne une autre technique pour obtenir la représentation des entiers négatifs. Méthode qui aboutit au même résultat évidemment. Rappel : pour les entiers positifs, il suffit de les écrire en binaire. Sur un mot de taille 8 en binaire, 12 est représenté par 0000 1100, cherchons à représenter -12 sur un mot de taille 8.

La méthode précédente dit qu'il faut calculer $-12+256$, soit 244 et $244 = 11110100_2$. Retrouvons ce résultat autrement.

Complément à deux (méthode)

Méthode : pour obtenir le complément à deux d'un entier négatif en binaire sur des mots de (taille) n bits ($n = 8$ ici), il suffit de :

1. Coder son opposé (positif) en binaire sur n bits ;
2. Inverser tous les bits ;
3. Ajouter 1 (addition binaire) à la ligne précédente.

Exemple : calculons le complément à deux de -12 sur 8 bits :

1. On code -(-12) soit 12 en binaire :		0	0	0	0	1	1	0	0
2. On inverse les bits :		1	1	1	1	0	0^1	1^1	1
3. Ajout de 1 :	+	0	0	0	0	0	0	0	1
Résultat (complément à deux) :	=	**1**	**1**	**1**	**1**	**0**	**1**	**0**	**0**

Conclusion : on retrouve 11110100, le complément à deux de -12 sur 8 bits.

Remarques :

1. Reprenons l'addition binaire ci-dessus colonne par colonne.
   Dernière colonne : $1_{(2)} + 1_{(2)} = 10_{(2)}$: on pose 0 et on retient 1.
   Avant-dernière colonne $1_{(2)^{(\text{retenue})}} + 1_{(2)} + 0_{(2)} = 10_{(2)}$: on pose 0 et on retient 1.
   Colonne précédente : $1_{(2)^{(\text{retenue})}} + 0_{(2)} + 0_{(2)} = 1_{(2)}$: on pose 1. Ainsi de suite.
2. Les nombres positifs sont finalement codés sous la forme $0b_1b_2b_3b_4b_5b_6b_7$ avec $b_i \in \{0,1\}$, les nombres négatifs de la forme $1b'_1b'_2b'_3b'_4b'_5b'_6b'_7$ avec $b'_i \in \{0,1\}$, par conséquent, **le premier bit donne le signe de l'entier**. Ainsi : les entiers positifs ont un premier bit égal à 0, les entiers négatifs ont un premier bit égal à 1 : le premier bit permet bien de connaître le signe de l'entier.

1.5 Entiers « longs » de Python

1.5.1 Introduction

La taille des mots, donc des nombres, généralement gérée par un PC est de 32 ou 64 bits. Le plus grand entier représentable sur un mot de taille fixe est donc $2^{64} - 1$. Mais **Python n'est pas limité pour les entiers** par la taille des mots. En effet Python connaît parmi ses types le type **entier long** :

```
[In] : type(2)       # quel est le type d'un entier («normal») ?
[Out]: <class 'int'> # un int
[In] : 2**200        # un entier long
[Out]: 1606938044258990275541962092341162602522202993782792835301376
[In] : type(2**200)  # quel est le type de ces entiers longs ?
[Out]: <class 'int'> # un int --> un seul type pour tous les entiers
```