Chapitre 12 :
Représentation des nombres entiers

I Généralités

1.1 Ecriture décimale

Nous avons l'habitude d'utiliser l'écriture décimale d'un nombre :

Proposition 1

Tout entier strictement positif s'écrit sous la forme :

$$x = a_r \times 10^r + a_{r-1} \times 10^{r-1} + ... + a_1 \times 10 + a_0$$

Dans cette écriture $r \in \mathbb{N}$, les a_i appartiennent à $[\![0,9]\!]$ et $a_r \neq 0$.

Pour obtenir l'écriture décimale d'un nombre entier, il suffit de considérer les restes des divisions euclidiennes successives par 10 :

- $237 = 23 \times 10 + ⑦$,
- $23 = 2 \times 10 + ③$,
- $2 = 0 \times 10 + ②$.

Ainsi l'écriture décimale est bien ②③⑦.

1.2 Ecriture binaire

En informatique, on utilise plutôt l'écriture en binaire (base 2) des nombres :

Proposition 2

Tout réel $x \in \mathbb{N}^*$ s'écrit :

$$x = 1 \times 2^r + a_{r-1} \times 2^{r-1} + ... + a_1 \times 2 + a_0$$

où $r \in \mathbb{Z}$, les a_i valent 0 ou 1.
Cette écriture est appelée écriture en base 2 de x.
L'écriture $\overline{1a_{r-1}...a_0}^2$ représentera le nombre $1 \times 2^r + a_{r-1} \times 2^{r-1} + ... + a_1 \times 2 + a_0$.

Remarque : L'écriture binaire est utilisée en informatique car on peut considérer qu'un circuit électronique n'a que 2 états qui sont alors représentés par les chiffres 0 et 1.

De même que pour l'écriture décimale, pour obtenir l'écriture binaire d'un nombre entier, il suffit de considérer les restes des divisions euclidiennes successives par 2 :

- $237 = 118 \times 2 + ①$,
- $118 = 59 \times 2 + ⓪$,
- $59 = 29 \times 2 + ①$,
- $29 = 14 \times 2 + ①$,
- $14 = 7 \times 2 + ⓪$,
- $7 = 3 \times 2 + ①$,
- $3 = 1 \times 2 + ①$,
- $1 = 0 \times 2 + ①$.

Ainsi l'écriture binaire de $\overline{11101101}^2$.

➪ **Exemple 1 :**

- Ecriture décimale de $a = \overline{101}^2$:

- Ecriture décimale de $b = \overline{1011}^2$:

- Ecriture binaire de $c = 22$:

- Ecriture binaire de $d = 125$:

1.3 Vocabulaire

On travaille dorénavant avec des représentations en binaire. On utilise alors le vocabulaire suivant :

- l'ensemble $\{0,1\}$ appelé l'**alphabet**,
- les chiffres 0 et 1 sont appelés des **lettres**,
- un **mot** est une suite composée de 0 et de 1.

La taille d'un mot s'exprime en **bits** et un mot de 8 bits est un **octet**.

II Représentation des entiers

2.1 Représentation des entiers positifs sur des mots de taille fixe

On va représenter un entier positif en utilisant des mots de n bits avec, en pratique, $n = 32$ ou $n = 64$ (processeurs les plus courants). On peut ainsi représenter tous les entiers compris entre 0 et $2^n - 1$.

Cette représentation est limitée car elle ne permet pas de travailler sur des nombres négatifs mais, de plus, elle fait perdre le bit de poids fort en cas de dépassement. Par exemple, en utilisant une représentation sur 4 bits, si on somme 14 et 8 :

$$14 + 8 = \overline{1110}^2 + \overline{1000}^2 = \overline{10110}^2$$

Comme la représentation est limitée à 4 bits, on perd le premier bit et on obtient alors :

$$14 + 8 = \overline{0110}^2 = 6.$$

2.2 Représentation des entiers signés sur des mots de taille fixe

On va représenter un entier signé (de signe quelconque) en utilisant des mots de n bits avec, en pratique, $n = 32$ ou $n = 64$. L'idée la plus naturelle est de réserver le premier bit afin de représenter le signe : 0 pour un signe + et 1 pour un signe -. Par exemple, comme $6 = \overline{110}^2$, sur 4 bits, +6 sera représenté par 0110 et -6 par 1110. Cette méthode donne alors 2 représentations de 0 : 0000 et 1000. De plus, l'inconvénient majeur de cette méthode est qu'il faut des algorithmes différents pour l'addition de nombres de même signe et l'addition de nombres de signes opposés.

Afin de remédier à ce problème, on utilisera la **représentation en complément à deux** :

- le premier bit représente le signe : 0 pour les entiers positifs et 1 pour les négatifs,
- les entiers positifs sont représentés de façon standard en utilisant les $n-1$ bits restants. On peut donc représenter les entiers entre 0 et $2^{n-1}-1$.
- soit m un entier strictement négatif. La représentation du complément à deux consiste à représenter m par la différence $2^n - (-m)$. En pratique :
  - on représente $-m$ (entier positif) en écriture binaire sur n bits, on inverse les bits (0 devient 1 et 1 devient 0) et on obtient un nombre m' tel que :

$$-m + m' = \overline{11\ldots11}^2 = \sum_{k=0}^{n-1} 2^k = \frac{1-2^n}{1-2} = 2^n - 1.$$

  - on rajoute 1 : $m'' = m' + 1$, on a alors :

$$-m + m'' = 2^n,$$

    d'où :

$$m'' = 2^n - (-m).$$

    Et comme sur n bits, $2^n = \overline{00\ldots00}^2$, m'' représente m.
- On représente donc les entiers entre -2^{n-1} et $2^{n-1}-1$.

Avec cette méthode, 0 n'a qu'une représentation et l'addition est plus simple à programmer.

➪ **Exemple 2 :**

- Représentation de -6 sur 4 bits :

- Représentation de -25 sur 6 bits :

- Nombre représenté, sur 4 bits, par 1001 :

- Nombre représenté, sur 6 bits, par 110101 :

2.3 Entiers multi-précision

La méthode du complément à deux permet de représenter les entiers entre $-2,147483648.10^9$ et $2,147483647.10^9$ sur 32 bits et entre $-9,223372036854775808.10^{18}$ et $9,223372036854775807.10^{18}$ sur 64 bits. Cependant cette représentation reste limitée.

Le langage Python utilise une représentation multi-précision des entiers, celle-ci est dite illimitée (elle n'est limitée que par la capacité de la mémoire). Le principe est le suivant : si un entier dépasse la taille maximale, ce nombre est découpé en plusieurs parties et c'est Python qui se chargera des opérations à effectuer. Le programme sera alors ralenti mais les entiers seront bien représentés.

La représentation multi-précision rend difficile l'évaluation du coût de chaque opération arithmétique car le coût varie en fonction de la taille de l'entier.