Cours II

Représentation des nombres en binaire

Lycée Louis Thuillier - Informatique - PCSIB -

Ce cours est largement inspiré du livre Toute l'informatique en CPGE scientifique de E. Cochard, S. Rainero, M. RoussangeE. Sebert-Cuvillier
Merci également à Paul Stien pour m'avoir aidé sur le côté "math" !

Table des matières

Un ordinateur ne manipule que des nombres : tout fichier (un texte, une image, un son, ...) est enregistré sous forme d'un codage bien précis de nombre. Il est alors primordial de comprendre comment un ordinateur se représente les nombres.

Objectif du cours :
▷ comprendre comment sont représentés les nombre
▷ **pour comprendre les problèmes qui peuvent subvenir** lorsqu'on utilise l'outil informatique.

On va construire pas à pas le codage des nombres en allant des plus simples au plus complexes :

entier naturel → entier relatif → nombre réel

1 Comment écrire une nombre ?

On cherche à transmettre sans parler ni écrire le nombre de grains de riz contenu et compté dans un paquet de 1kg. On à compter $N = 102\,364$ grains de riz. Comment faire ?

Le nombre N ainsi écrit est un enchaînement de chiffres qui signifient le nombre de grains de riz est égale à :

$$N = 1 \times 10^5 + 0 \times 10^4 + 2 \times 10^3 + 3 \times 10^2 + 6 \times 10^1 + 4 \times 10^0$$

On a, par convention, choisi un système décimal (du fait du nombre de doigts sur nos mains). On remarque alors que pour **décrire** un nombre en système décimale il suffit d'avoir un symbole pour 10 chiffres (0,1,2,3,4,5,6,7,8,9).

1.1 Entier naturel

Écriture un nombre dans une base

Soit un entier b, qui nous servira de base. Tout entier naturel n non nul peut s'écrire de manière unique comme :

$$n = a_p b^p + a_{p-1} b^{p-1} + ... + a_0 b^0$$

où chacun des a_i est compris entre 0 et $b-1$: $0 \leq a_i \leq b$. On dit alors que $\overline{a_p a_{p-1} ... a_0}_b$ est le **développement** de n **en base** b.

Pour écrire n'importe quel nombre entier n, il faut pouvoir représenter les nombres a_i qui apparaissent dans le développement de n.

| ***Exemple 1 :*** *Le nombre* $N = 102\,364$ *est l'écriture en base* 10 *du nombre* $102\,364 = \overline{102364}_1 0$.

Problème : la mémoire d'une machine numérique n'est constitué que d'éléments possédant deux états :

▷ CD : surfaces réfléchissantes ou non réfléchissantes
▷ carte SIM : aimants orientés vers le haut ou le bas
▷ SSD : supraconducteurs qui laisse passer ou ne laisse pas passer le courant

On généralise ces états par deux états

$$\uparrow \text{ : "up" ou 1 et } \downarrow \text{ : "down" ou 0}$$

Ces constituant élémentaires sont appelés des bits, forme contracté de "binary digits".
Conséquence : comme les a_i ne peuvent valoir que $a_i = 0$ ou $a_i = 1 \Rightarrow$ on utilise la base 2.

Propriété. Ecriture en base 2
tout entier naturel n non nul peut s'écrire de manière unique comme :

$$n = a_p 2^p + a_{p-1} 2^{p-1} + ... + a_0 2^0 \tag{1.1}$$

où chacun des a_i est égale soit à 0 soit à 1 pour $0 < i < p-1$ et $a_p = 1$.

Un nombre entier s'écrit alors : $\overline{a_1 a_2 a_3 ...}$, c'est ce qu'on appelle un mot machine.

$$\overline{001101} = 2^0 + 2^2 + 2^3$$

► Important ! Taille des nombres représentés Important !

On rencontre souvent deux versions d'un même programme : un codé en 32bits, l'autre en 64bits. Quelle différences ?

Chaque bit permet de coder 0 ou 1, noté a_i, dans l'écriture en base 2 du nombre. Avec un octet, soit 8 bits, j'ai donc $\overline{a_1 a_2 a_3 a_4 a_5 a_6 a_6 a_8}$
Le nombre le plus grand qu'il,est possible d'écrire est $\overline{11111111}$ qui correspond en base 10 à :

$$2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = \frac{1-2^8}{1-2} = 2^8 - 1 = 255.$$

On retrouve par exemple les niveaux de couleurs dans une image : un entier n entre 0 et 255. Les couleurs des images sont codés en 8bits.

| ***Application 1 :*** *Trouver l'ensemble des nombres que l'on peut écrire avec* 16 *bits,* 64 *bits.*

Propriété. Nombre maximal représentable à N bits
Avec N bits, on peut représenter tous les nombres entiers entre 0 et $2^N - 1$.

| *Application 2 : Quelle est la parité d'un nombre dont l'écriture binaire se termine par un 1 ?*

1.2 Entiers relatifs

▶ Méthode naïve

Cette méthode ne fonctionne pas mais permet de comprendre le principe ...

Pour représenter les entiers relatifs, il est nécessaire d'utiliser un bit supplémentaire pour coder le signe du nombre. On ajoute alors en début de l'écriture un bit qui précise le signe du nombre, 0 pour un entier positif et 1 pour un entier négatif.
Pour différentes raisons, cette méthode ne fonctionne pas. Par exemple le nombre 0 peut être codé de deux façon différentes
- ▷ un 1 suivi de 0
- ▷ un 0 suivi de 0

▶ Méthode du complément à deux

Pour remédier à cela, et conserver la représentation décrite précédemment des nombres entier, on utilise la **notation en complément à deux**.

Propriété. Complément à deux
En utilisant N bits, la représentation d'un entier strictement négatif n est celle de $n + 2^N$.

Exemple 2 : Donner la représentation de -6 avec 8 bits.

- ▷ *Pour représenter -6, on va chercher à coder $2^8 - 6 = 250$*
- ▷ $250 = 2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7$ *donc* $\overline{11111010}$

Mais si je veux représenter 250 ? C'est impossible, la place a été prise par -6.

Schéma de la méthode

Propriété. Limite à N bits
Une représentation à N bit peut représenter tous les entiers n compris entre :

$$-2^{N-1} \leq n \leq 2^{N-1} - 1$$

Attention ! c'est puissance $N - 1$!! On a coupé en deux l'intervalle des nombres représentables (la moitié pour les > 0 et la moitié pour les < 0).

Pourquoi complément à deux ?
Pour écrire $-n$
- ▷ on écrit n en base 2
- ▷ puis on transforme tous les 1 en 0 et les 0 en 1
- ▷ on ajoute 1 à la fin

1.3 Entier multi-précision

$2^{8-1} = 128$ et $2^{16-1} = 32768$ ... Or en Python je peux manipuler des nombres bien plus grand. Comment est-ce possible ?

Propriété. Entier multi-précision
En Python, la taille des entiers n'est pas limitée : le nombre N de bits utilisé pour coder les nombres entier n'est pas prédéfini et s'adapte au nombre manipulé.
Ce sont les entiers multi-précision.

Récapitulatif : représenter les nombres entiers

▷ **entier naturels :**
si le nombre que je manipule est un entier positif, je peux représenter tous les nombres compris entre 0 et $2^N - 1$.

▷ **entiers relatifs, méthode du complément à deux :**
le nombre N de bits est fixé ⇒ la taille des nombres représentés est limitée ($-2^{N-1} \to 2^{N-1}$) **mais ça va plus vite**

▷ **entiers relatifs, méthode multi-précision :**
le nombre N de bits est variables et s'adapte ⇒ la taille des nombres représentés n'est pas limitée **mais c'est leeeeent à manipuler**

2 Représentation des nombres réels

Objectifs
1. comprendre pourquoi il existe une limite maximale à la représentation des nombres
2. comprendre pourquoi on fait tout le temps des approximations lorsqu'on représente un réel

2.1 Principe de l'écriture

Pour coder les nombres réels, on va s'inspirer de la façon dont on a représenté les nombres entiers. Une fois n'est pas coutume, on va faire *comme en physique ...*

Écriture scientifique :
le nombre 3543 s'écrit $3,543 \times 10^3$. On a alors besoin d'écrire :
▷ le signe, ici "+" sous-entendu
▷ le nombre "devant", compris entre 0 et 10
▷ l'exposant qui est un entier
Plutôt qu'en base 10, on va utiliser la base 2.

Définition. Nombre à virgule flottant en base 2
Tout nombre réel peut s'écrire comme : $s \times m \times 2^e$.
▷ **s** le signe
▷ **m** la mantisse avec $0 < m < 2$.
▷ **e** l'exposant, qui est un entier

La virgule est qualifiée de flottante car on peut la déplacer en faisant varier l'exposant. Les nombres ainsi représentés sont des nombres **flottant**, *float* en anglais. D'où le nom en Python ...

Il faut alors coder :
▷ le signe : 1bit **OK**
▷ la mantisse : un nombre réel entre 0 et 2
▷ l'exposant : un nombre entier

Exposant décalé

Pour coder l'exposant, on aurait pu utiliser la méthode du complément à deux. Pour des raisons qu'on ne détaillera pas ici, ce n'est pas pratique : on utilise la méthode de l'exposant décalé.

Propriété. Exposant décalé
Avec Nbits, pour représenter l'exposant réel e, on code l'exposant décalé m : $m = e + \left(2^{N-1} - 1\right)$.
On ajoute le **décalage** $\left(2^{N-1} - 1\right)$.

Contrairement à la méthode du complément à deux, on décale ici tous les entiers (et pas juste les entiers négatifs).

Schéma de la méthode

2.2 Représenter un nombre entre 0 et 2

Inspirons nous toujours de l'écriture des nombres entiers.

Propriété. Ecrire un nombre réel en base 2
Soit x un entier entre 0 et 2. On peut écrire alors :

$$x = \sum_{i=1} a_i 2^{-i} = a_1 2^{-1} + a_2 2^{-2} + a_3 2^{-3} + ...$$

avec $a_i = 0$ ou 1. On note alors $x = \overline{0, a_1 a_2 a_3 ....}$.

Attention ! Ca ressemble à l'écriture en base 2 d'un entier **MAIS** :
- ▷ on somme les inverses des puissances de 2
- ▷ **et surtout** la somme est potentiellement infini

> ***Exemple 3 :*** *Ecrire en base* 2 *et avec* 8*bits les nombres* 0.625 *et* 0.4
>
> - ▷ $0.625 = 0.5 + 0.125 = 2^{-1} + 2^{-3}$ *donc* $\overline{10100000}$
> - ▷ $0.4 = 0.25 + 0.125 + 0.015625 + ... = 2^{-2} + 2^{-3} + 2^{-6} + 2^{-7} + ...$ *: la décomposition est sans fin. C'est impossible.*
>
> *C'est le même problème qu'on rencontre en base* 10 *pour écrire* $1/3 = 0.333333....$ *!*

Propriété.
Il n'est pas possible de représenter tous les réel, ni tous les décimaux en base 2 : il n'est donc pas possible de coder les réel en informatique.

2.3 Mantisse et précision

On code une mantisse m (nombre entre 0 et 2) avec N bits. On écrit alors m comme :

$$m = \sum_{i=1}^{N} a_i 2^{-i} = a_1 2^{-1} + a_2 2^{-2} + a_3 2^{-3} + ... + a_N 2^{-N}$$

On perd alors la précision du terme $2^{-(N+1)}$.

Propriété. Mantisse et précision
En utilisant N bits pour coder une mantisse, deux nombres distants de $2^{-(N+1)}$ sont indiscernables.

Il faut alors allouer le maximum de bits pour la mantisse. Problème, le nombre total de bits disponibles pour coder un réel est fixé.

▶ Quelques exemples

Le nombre $-3141,5$ en 32 bits. Il existe différentes répartitions des bits mais on prendra un cas *classique*.

- ▷ 1 bit pour le signe
- ▷ 8 bits pour l'exposant
- ▷ 23 bits pour la mantisse.

Nombre maximale et précision

▷ **Nombre maximal**
l'exposant varie entre -2^{8-1} et 2^{8-1} soit entre -128 et 128. Donc le maximum est $2^{+128} \simeq 3\,10^{38}$
Si l'on essaie de représenter un nombre plus grand que 10^{38}, on obtient un débordement par valeur supérieure (on appelle cela un *overflow*). Typiquement, cela arrive si on additionne deux nombres dont la somme dépasse la limite.

▷ **précision** :
avec 23 bits pour la mantisse, la précision est $2^{-(23+1)} \simeq 6\ 10^{-8}$. Donc $7\ 10^{-9}$ et $3,2\ 10^{-12}$ sont deux mêmes nombres dans notre ordinateur.

Schéma de l'écriture

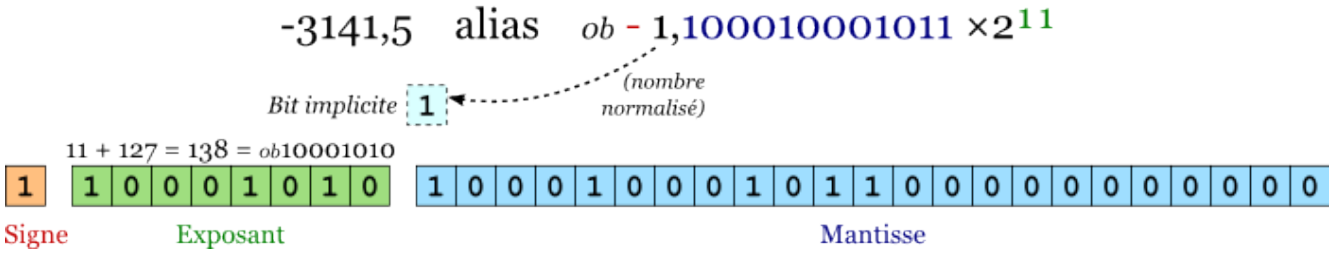

Fig. 1 – Ecriture en simple précision du nombre $-3141,5$.

Exemple 4 : On cherche à l'écriture décimal de $X = \overline{10001001001001001101001110110000}$.

1. *le premier bit est consacré au signe :* $1 \rightarrow$ " $-$ " *donc c'est un nombre négatif*
2. *les 8 bits qui suivent sont associés à l'exposant décalé :* $d = \overline{00010010}$. *On a donc :*

$$d = 2 + 16 = 18.$$

Donc $e = 18 - 127 = -109$.

3. *Les 23 derniers bits sont dédiés à la mantisse tout en sachant que le* 1 *devant la virgule n'est pas codé. On a donc :*

$$\overline{01001001101001110110000} \rightarrow \overline{1,01001001101001110110000},$$

d'où $m = \overline{1,01001001101001110110 00}$ *:*

$$m = 1\times2^{0}+1\times2^{-2}+0\times2^{-3}+1\times2^{-5}+1\times2^{-8}+1\times2^{-9}+1\times2^{-11}+1\times2^{-14}+1\times2^{-15}+1\times2^{-16}+1\times2^{-18}+1\times2^{-19}$$

$$m = 1.287710189819336$$

4. *On a donc :*

$$X = -\,1.287710189819336 \times 2^{-109} = -\,1.9840316914125045 \times 10^{-33}$$

3 Conséquence en Python

Nous allons décrire ici, sans chercher à expliquer, les conséquences pratiques de la représentation des nombres flottant.

Idée fondamentale : les principales sources d'erreur dues à la représentation des nombres en informatique provient du nombre fini de bit utilisés pour coder ces nombres.

▷ taille finie de la mantisse ⇒ nombre fini de chiffres après la virgule ⇒ approximation des nombres réels
▷ taille finie de l'exposant ⇒ plus grand nombre/plus petit nombre possible.

Exemple 5 :

▷ *Opération et arrondi*

```
>>> 0.2+0.2-0.4
0.0

>>> 0.1+0.1+0.1-0.3
5.551115123125783e-17

>>> 0.1+0.1+0.1-0.3==0
False
```

Méthode en DS. Règle numéro uno
ne jamais faire de test d'égalité strict sur des réels ! !, on vérifiera toujours les égalités à une précision δx près.

▷ *Nombre flottant trop grand*

```
>>> 2**1025
359538626972463181545861038157804946723595395788461314546860162315465351611001926265416954644815072042240227759742786715317579537628833244985694861278948248755535786849730970552604439202492188238906165904170011537676301364684925762947826221081654474326701021369172596479894491876959432609670712659248448274432

>>> 2.0**1024
Traceback (most recent call last):
  File "<console>", line 1, in <module>
OverflowError: (34, 'Result too large')
```

Il n'y a pas de problème avec les entiers car ils sont codés en muli-précision : le nombre de bit s'adapte à la taille du chiffre.
⇒ ***Attention ! quand on manipule des grands nombres flottants***

Méthode en DS. Règle numéro dos
on réfléchira toujours à l'avance la valeur maximale que peut prendre un nombre réel dans un programme

Exemple 6 :

Il n'est pas catastrophique de faire ces erreurs ... sauf si vous êtes concepteur de fusée !
L'explosion de la fusée Ariane 5 (Juin 1996, 500 millions de dollars) : le prototype de la fusée Ariane 5 explose 40 secondes après son décollage à cause d'une erreur de type overflow. Un nombre à virgule flottante codé sur 64 bits désignant la vitesse horizontale en m/s de la fusée se retrouve converti par erreur en entier codé sur 16 bits dans le programme. Durant le décollage, la vitesse dépasse les 32767 m/s (plus grand nombre entier codable en 16 bits). Ceci provoque un overflow : le nombre est trop grand pour être codé. Le programme plante, la fusée dévie de sa trajectoire puis explose.