CHAPITRE 2

Représentation des nombres

Jérôme VON BUHREN
http://vonbuhren.free.fr

CPGE - Filière Scientifique - 1re année

L'unité de mémoire en informatique est le **bit** (physiquement constitué d'un composant électronique pouvant prendre deux états que l'on note 0 et 1). Par conséquent, pour sauvegarder un objet dans la mémoire d'un ordinateur, il faut donc parvenir à le représenter uniquement à l'aide de 0 et de 1.

En pratique, la mémoire des ordinateurs est découpée en blocs de 8 bits (un **octet**) et un ordinateur avec un processeur de 64 bits manipule des paquets de 8 octets.

Dans ce chapitre, nous allons étudier la représentation de différents types de nombres dans la mémoire des ordinateurs.

Nous sommes tous habitués depuis l'enfance à utiliser l'écriture en base 10 des entiers naturels. Par exemple, l'écriture 8532 représente le nombre

$$8 \times 10^3 + 5 \times 10^2 + 3 \times 10^1 + 2 \times 10^0.$$

Pour ce choix classique, les nombres $0, \ldots, 9$ sont appelés les chiffres (de la base 10).

Plus généralement, on peut remplacer 10 par tout entier $b \in \mathbb{N} \setminus \{0,1\}$.

Théorème (Représentation d'un entier naturel en base b)

Soit $b \in \mathbb{N} \setminus \{0,1\}$. Pour tout nombre $x \in \mathbb{N}$, il existe un unique entier $p \in \mathbb{N}$ et un unique p-uplet $(x_0, \dots, x_p) \in [\![0, b-1]\!]^{p+1}$ tel que $x_p \neq 0$ et

$$x = x_p b^p + x_{p-1} b^{p-1} + \cdots + x_1 b + x_0 = \sum_{k=0}^{p} x_k b^k.$$

Remarques 1

a) Le p-uplet $(x_0, \dots, x_p) \in [\![0, b-1]\!]^{p+1}$ est appelé la représentation en base b de l'entier x.

b) Les éléments $0, 1, \dots, b-1$ sont appelés les chiffres de la base b.

c) Pour tout $p \in \mathbb{N}$ et $(x_0, \dots, x_p) \in [\![0, b-1]\!]^{p+1}$,
on note $\left(x_p x_{p-1} \dots x_0\right)_b = \sum_{k=0}^{p} x_k b^k$.

d) Dans le cadre de l'informatique, nous utiliserons principalement la décomposition en base 2 (écriture binaire) dont les chiffres sont 0 et 1.

Exemple 1

On a $\left(11011\right)_2 = 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 27$.

Remarque 2

Pour déterminer la décomposition binaire d'un entier en binaire, on peut effectuer une succession de divisions euclidienne par 2.

Exemple 2

On souhaite décomposer 57 en binaire. On effectue une succession de divisions euclidienne.

$$\begin{array}{rcrcrcr} 57 & = & 2 & \times & 28 & + & \mathbf{1} \\ 28 & = & 2 & \times & 14 & + & \mathbf{0} \\ 14 & = & 2 & \times & 7 & + & \mathbf{0} \\ 7 & = & 2 & \times & 3 & + & \mathbf{1} \\ 3 & = & 2 & \times & 1 & + & \mathbf{1} \\ 1 & = & 2 & \times & 0 & + & \mathbf{1}. \end{array}$$

On en déduit que la représentation binaire de 57 est $(111001)_2$.

Exercice 1

Déterminer la représentation binaire de 124.

Exercice 2

Écrire une fonction `base_2_vers_10(rep:str) -> int` prenant en entrée une chaîne de caractères `rep` constituée de `0` et de `1` et renvoyant le nombre représenté par `rep` en binaire. Voici un exemple d'exécution de la fonction.

```
>>> base_2_vers_10('111001')
57
```

Exercice 3

Écrire une fonction `base_10_vers_2(n:int) -> str` prenant en entrée un entier naturel `n` et renvoyant la représentation binaire de `n`. Voici un exemple d'exécution de la fonction.

```
>>> base_10_vers_2(57)
'111001'
```

En binaire, comme les seuls chiffres sont 0 et 1, les opérations usuelles (addition, soustraction, multiplication, division) sont particulièrement simples à réaliser. Il suffit d'utiliser les relations (en binaire) suivantes

$$0+0=0, \qquad 0+1=1, \qquad 1+0=1 \qquad 1+1=10$$

et d'adapter à la base 2 les algorithmes de calcul appris à l'école primaire (pour la base 10).

Exemple 3

Voici quelques exemples d'opérations effectuées en binaire.

$$\begin{array}{cccccccc} & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ + & & 1 & 1 & 1 & 1 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \end{array}$$

$$\begin{array}{cccccccc} & 1 & 0 & 0 & 1 & 1 & 0 & 1 \\ - & & 1 & 1 & 1 & 1 & 0 & 1 \\ \hline & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}$$

$$\begin{array}{cccccc} & & 1 & 0 & 1 & 0 \\ \times & & & 1 & 0 & 1 \\ \hline & & 1 & 0 & 1 & 0 \\ & 0 & 0 & 0 & 0 & \cdot \\ 1 & 0 & 1 & 0 & \cdot & \cdot \\ \hline 1 & 1 & 0 & 0 & 1 & 0 \end{array}$$

Exercice 4

Réaliser les opérations ci-dessous en binaire comme dans l'exemple ci-dessus.

$$(101010)_2 + (11011)_2, \quad (110111)_2 - (11011)_2,$$

$$(10101)_2 \times (1011)_2, \quad (1100101)_2 \div (1011)_2.$$

Pour sauvegarder un entier naturel dans la mémoire de l'ordinateur, on utilise sa représentation en binaire. Sur une architecture de n bits, ce procédé permet de représenter les 2^n entiers naturel compris entre

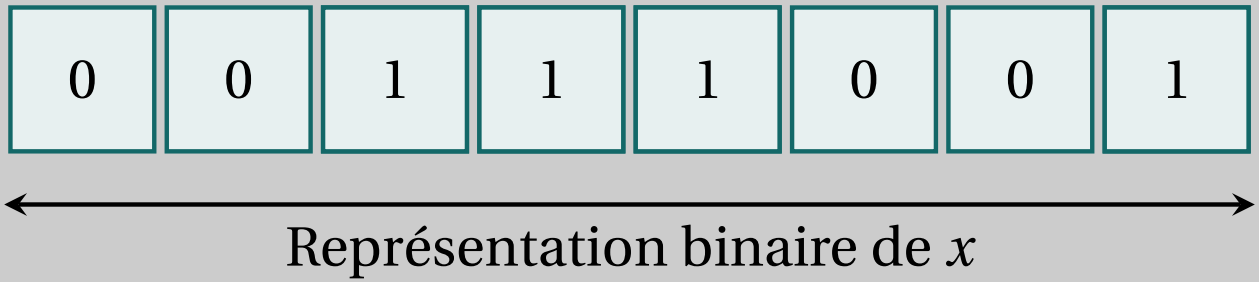
$$0 = \big(\underbrace{0\ldots\ldots0}_{n \text{ fois}}\big)_2 \quad \text{et} \quad 2^n - 1 = \big(\underbrace{1\ldots\ldots1}_{n \text{ fois}}\big)_2.$$

En particulier, avec un ordinateur admettant une architecture classique à 64 bits (ou 8 octets), on peut représenter les entiers naturels compris entre

$$0 \quad \text{et} \quad 2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615.$$

Exemple 4

L'entier naturel $x = 57 = (111001)_2$ est représenté sur une architecture de 8 bits comme ci-dessous.



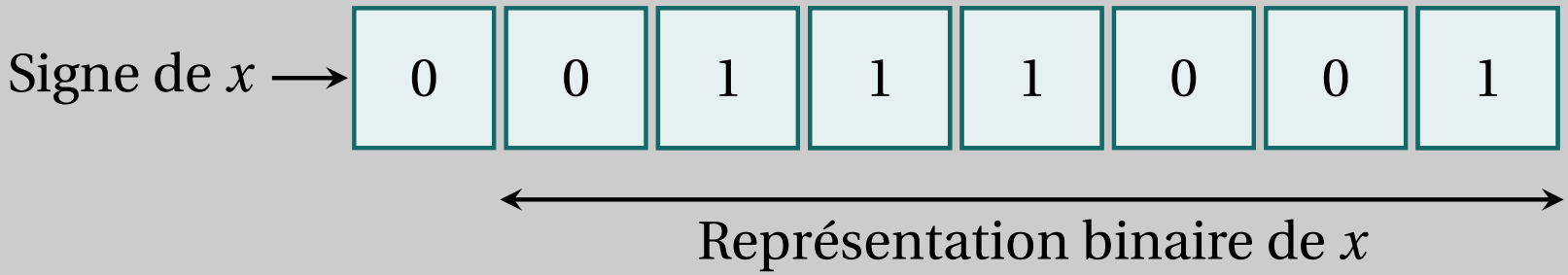
Pour représenter un entier relatif, l'usage est de coder son signe sur le premier bit (0 pour les entiers positifs et 1 pour les entiers négatifs) et de réserver les bits restants pour coder l'entier.

Pour coder un entier relatif positif x sur n bits, on complète le 0 initial (indiquant le signe) par la représentation binaire de x. On en déduit que le plus grand entier relatif positif représentable sur n bits est $2^{n-1}-1$. En particulier, avec un ordinateur admettant une architecture classique à 64 bits (ou 8 octets), le plus grand entier relatif positif représentable est

$$2^{63}-1 = 9\,223\,372\,036\,854\,775\,807.$$

Exemple 5

L'entier relatif $x = 57 = (111001)_2$ est représenté sur une architecture de 8 bits comme ci-dessous.



Pour coder un entier relatif négatif x sur n bits, on pourrait compléter le 1 initial (indiquant le signe) par la représentation binaire de $|x|$. Cependant, cette approche présenterait plusieurs inconvénients :

(i) le nombre $0 = +0 = -0$ aurait deux représentations distinctes (ce qui n'est pas satisfaisant car il est toujours préférable d'avoir une unique représentation pour un objet) ;

(ii) l'algorithme d'addition de deux nombres binaires vue dans la partie précédente ne s'appliquerait que à des entiers de même signe.

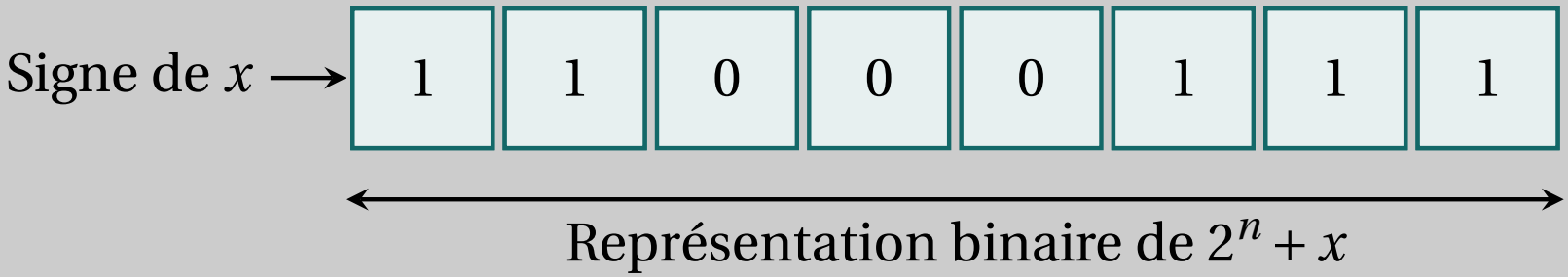
Pour pallier ces difficultés, nous allons utiliser la méthode du complément à deux : le nombre relatif négatif x est représenté en mémoire sur n bits par la représentation binaire de l'entier positif $2^n + x$. Pour que le premier bit (qui représente le signe) soit bien égal à 1, il est nécessaire que

$$2^{n-1} \leqslant 2^n + x < 2^n \quad \Leftrightarrow \quad -2^{n-1} \leqslant x < 0.$$

On en déduit que le plus petit entier relatif positif représentable sur n bits est -2^{n-1}.

Exemple 6

L'entier relatif $x = -57$ est représenté sur une architecture de 8 bits par la décomposition binaire du nombre $2^8 - 57 = 199 = (11000111)_2$.



Bilan

Les entiers relatifs représentables par la méthode du complément à deux sur une architecture à n bits sont ceux compris entre -2^{n-1} et $2^{n-1}-1$.

- Si x est un entier vérifiant $0 \leqslant x \leqslant 2^{n-1}-1$, il est encodé avec sa représentation binaire.
- Si x est un entier vérifiant $-2^{n-1} \leqslant x < 0$, il est encodé avec la représentation binaire de $2^n + x$.

Ces choix permettent d'éviter les inconvénients évoqués précédemment :

(i) tout entier relatif de $\Omega = [\![-2^{n-1}, 2^{n-1}-1]\!]$ admet une unique représentation.

(ii) on peut vérifier que si x et y sont deux éléments de Ω tels que $x+y \in \Omega$, alors la représentation de $x+y$ s'obtient en ne gardant que les n bits de poids faibles de la somme des représentations de x et y.

Remarque 3

En appliquant le procédé de représentation décrit précédemment, on obtient le tableau ci-dessous.

Entier relatif	Représentation en mémoire
0	0 0 0 0 ----- 0 0 0 0
1	0 0 0 0 ----- 0 0 0 1
$\vdots$	-----
$2^{n-1}-1$	0 1 1 1 ----- 1 1 1 1
-2^{n-1}	1 0 0 0 ----- 0 0 0 0
$-2^{n-1}+1$	1 0 0 0 ----- 0 0 0 1
$\vdots$	-----
-2	1 1 1 1 ----- 1 1 1 0
-1	1 1 1 1 ----- 1 1 1 1

Exercice 5

Dans une représentation en complément à deux sur 8 bits, quels sont les entiers relatifs représentés par `01110101` et par `10010110` ?

Exemple 7

Effectuons une addition de deux entiers relatifs représentés par la méthode du complément à deux sur une architecture à $n = 8$ bits. Dans ce cadre, on peut représenter les entiers de $[\![-2^7, 2^7 - 1]\!] = [\![-128, 127]\!]$.

L'entier $x = 102 = (1100110)_2$ est positif, donc il est représenté par `01100110`. L'entier $y = -57$ est négatif, donc il est représenté par l'écriture binaire de $-57 + 2^8 = 199 = (11000111)_2$, i.e. `11000111`. En additionnant les représentations respectives de x et y, on obtient

$$\begin{array}{cccccccccc} & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ + & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ \hline 1 & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} \end{array}$$

En ne conservant que les 8 bits de poids faibles, on obtient la représentation `00101101`. Le premier bit étant un 0, la somme est positive et elle représente $(101101)_2 = 45$ qui est bien la somme de $x = 102$ et $y = -57$.

Nous venons de voir que les entiers relatifs représentables par la méthode du complément à deux sur une architecture à n bits sont ceux de l'intervalle $[\![-2^{n-1}, 2^{n-1}-1]\!]$. En général, si on effectue des opérations sur ces entiers, il est possible que le résultat ne soit plus un élément de cet ensemble : ce phénomène est appelé un **dépassement d'entier** ou un **débordement d'entier**.

En cas de dépassement d'entier, le résultat renvoyé par l'ordinateur dépend du langage utilisé. Dans la suite, on se limite à étudier le cas de Python. Ce dernier code les entiers signés sur $n = 64$ bits, donc le plus grand entier relatif positif représentable sur 64 bits en Python est

$$2^{63} - 1 = 9\,223\,372\,036\,854\,775\,807.$$

Provoquons volontairement un dépassement d'entier pour observer le comportement de Python.

```
>>> x = 9223372036854775807
>>> x + 1
9223372036854775808
```

En fait, l'interprète Python détecte automatiquement le dépassement d'entier : lorsque le résultat d'une opération sur des entiers sort de l'intervalle $[\![-2^{63}, 2^{63}-1]\!]$, la représentation des entiers par complément à deux est remplacée par une nouvelle forme : celle des **entiers multi-précisions**. Nous n'étudierons pas en détail la représentation en mémoire de cette dernière. Précisons simplement qu'elle a l'avantage de ne pas être limitée en taille (sauf par l'espace mémoire disponible sur la machine), mais elle présente aussi des inconvénients : les opérations arithmétiques sur des entiers multi-précisions sont plus lentes.

Remarque 4

En particulier, lorsqu'on manipule des entiers multi-précisions, qui peuvent être de taille arbitrairement grande, on ne peut plus considérer que les opérations arithmétiques sur ces nombres ont une complexité temporelle en $O(1)$.

De la même manière que nous manipulons usuellement l'écriture en base 10 des entiers naturels, nous sommes habitués à utiliser les nombres décimaux. Rappelons qu'un nombre **décimal** est un réel qui peut s'écrire sous la forme $x = a \times 10^{-n}$ avec $a \in \mathbb{Z}$ et $n \in \mathbb{N}$. En décomposant $|a|$ dans la base 10, on en déduit que l'on peut écrire

$$x = \pm \left(\sum_{k=0}^{p} x_k 10^k + \sum_{k=1}^{q} d_k 10^{-k} \right)$$

où $(p, q) \in \mathbb{N} \times \mathbb{N}^*$ et $(x_0, \dots, x_p, d_1, \dots d_q) \in [\![0, 9]\!]^{p+q+1}$.

Notation

Pour tout $(p, q) \in \mathbb{N} \times \mathbb{N}^*$ et $(x_0, \dots, x_p, d_1, \dots d_q) \in [\![0, 9]\!]^{p+q+1}$, on notera

$$\left(x_p x_{p-1} \dots x_0, d_1 \dots d_q\right)_{10} = \sum_{k=0}^{p} x_k 10^k + \sum_{k=1}^{q} d_k 10^{-k}.$$

Exemple 8

On a les égalités

$$\begin{aligned} 12,345 = \frac{12345}{10^3} &= \frac{1\times 10^4 + 2\times 10^3 + 3\times 10^2 + 4\times 10^1 + 5\times 10^0}{10^3} \\ &= 1\times 10^1 + 2\times 10^0 + 3\times 10^{-1} + 4\times 10^{-2} + 5\times 10^{-3} \\ &= (12,345)_{10}. \end{aligned}$$

De manière analogue, on peut considérer les nombres **dyadiques** : ce sont les nombres réels de la forme $x = a \times 2^{-n}$ avec $a \in \mathbb{Z}$ et $n \in \mathbb{N}$. En décomposant $|a|$ dans la base 2, on en déduit que l'on peut écrire

$$x = \pm \left(\sum_{k=0}^{p} x_k 2^k + \sum_{k=1}^{q} d_k 2^{-k} \right)$$

où $(p, q) \in \mathbb{N} \times \mathbb{N}^*$ et $(x_0, \dots, x_p, d_1, \dots d_q) \in [\![0, 1]\!]^{p+q+1}$. Par exemple, on a

Notation

Pour tout $(p, q) \in \mathbb{N} \times \mathbb{N}^*$ et $(x_0, \dots, x_p, d_1, \dots d_q) \in [\![0, 1]\!]^{p+q+1}$, on notera

$$\left(x_p x_{p-1} \dots x_0, d_1 \dots d_q\right)_2 = \sum_{k=0}^{p} x_k 2^k + \sum_{k=1}^{q} d_k 2^{-k}.$$

Exemple 9

On a les égalités

$$\begin{aligned} 1,625 = \frac{13}{8} &= \frac{1\times 2^3 + 1\times 2^2 + 0\times 2^1 + 1\times 2^0}{2^3} \\ &= 1\times 2^0 + 1\times 2^{-1} + 0\times 2^{-2} + 1\times 2^{-3} \\ &= (1,101)_2. \end{aligned}$$

Remarques 5

a) Tout nombre dyadique est un nombre décimal.

b) La réciproque est fausse : le nombre $0,1 = 10^{-1}$ est décimal, mais il n'est pas dyadique. On peut vérifier qu'on a le développement infini suivant

$$0,1 = \left(0,0\,0011\,0011\,0011\,0011\,0011\,0011\ldots\right)_2.$$

c) Par contre, on peut approcher tout nombre réel $x \in \mathbb{R}$ par un nombre dyadique avec une précision arbitrairement grande. En effet, pour tout $n \in \mathbb{N}$, en notant $a = \lfloor 2^n x \rfloor$, on a

$$2^n x - 1 < a \leqslant 2^n x \quad \Leftrightarrow \quad 0 \leqslant x - a \times 2^{-n} < 2^{-n}.$$

Exemple 10

En appliquant la remarque précédente pour $x = 0,1$ et $n = 10$, on obtient $a = \lfloor 2^n x \rfloor = 102$ et on en déduit qu'une approximation à 2^{-10} près de x par un nombre dyadique est

$$102 \times 2^{-10} = \left(0,000110011\right)_2.$$

Tout nombre dyadique non nul x peut s'écrire avec une **virgule flottante**, i.e. sous la forme

$$x = \pm 2^e \times \left(1, \underbrace{b_1 \dots b_r}_{m}\right)_2$$

où $e \in \mathbb{Z}$ et $(b_1, \dots, b_r) \in [\![0,1]\!]^r$. L'entier e est appelé l'**exposant** de x et m est appelé la **mantisse** de x.

Exemple 11

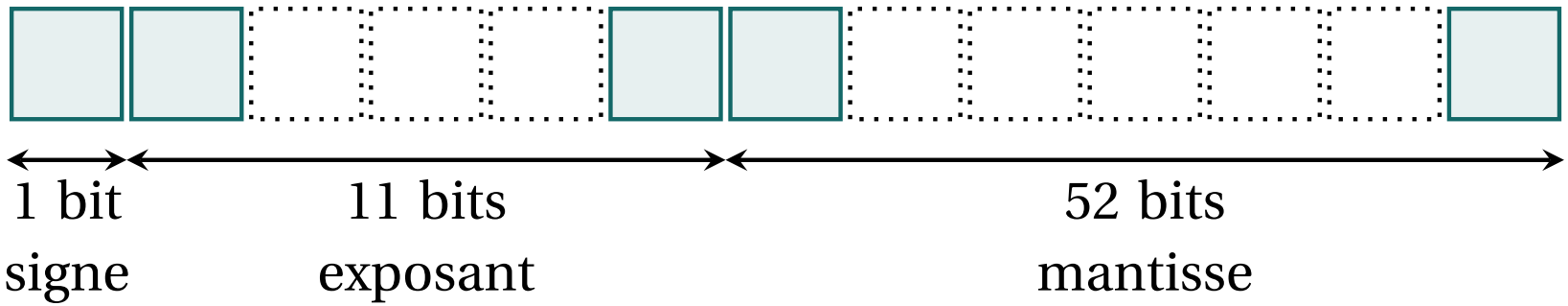
L'écriture de $x = 102 \times 2^{-10}$ avec une virgule flottante est

$$x = \left(0,000110011\right)_2 = +2^{-4} \times \left(1,10011\right)_2.$$

L'exposant de x est $e = -4$ et sa mantisse est $m = \left(10011\right)_2$.

Pour représenter les nombres dyadiques en mémoire, on utilise la norme IEEE-754 que nous allons présenter succinctement sur une architecture à 64 bits.

Dans cette norme, les nombres dyadiques sont codés sur 64 bits en mémoire : 1 bit pour le signe, 11 bits pour l'exposant et 52 bits pour la mantisse.



Donnons quelques précisions sur l'encodage de chacune des trois composantes.

- Le bit de signe vaut 0 pour un nombre positif et 1 pour un nombre négatif.
- L'exposant est codé avec la technique de décalage : l'exposant e est représenté par la décomposition binaire de $e' = e + 2^{10} - 1$. Comme on dispose de 11 bits pour stocker les chiffres de $e' = e + 2^{10} - 1$ en base 2, on a

$$0 \leqslant e' \leqslant 2^{11} \quad \Leftrightarrow \quad 1 - 2^{10} \leqslant e \leqslant 2^{10} \quad \Leftrightarrow \quad -1023 \leqslant e \leqslant 1024.$$

- Les 52 bits restants permettent de mémoriser les 52 premiers bits de la mantisse (quitte à ajouter des 0 si la mantisse à moins de 52 bits).

Remarques 6

En réalité la situation est un peu plus complexe.

a) Si l'exposant vérifie $-1022 \leqslant e \leqslant 1023$, alors on utilise la description précédente : on dit que ces nombres sont représentés sous forme **normalisée**.

b) Les encodages correspondant à $e = -1023$, ce qui équivaut à $e' = 0 = (00000000000)_2$, sont utilisés pour représenter les nombres de la forme $\pm 2^{-1022} \times (0, b_1 \dots b_{52})_2$. On dit que ces nombres sont représentés sous forme **dénormalisée**. En particulier, le nombre 0 est codé par 64 bits nuls en mémoire.

Remarques 6

En réalité la situation est un peu plus complexe.

c) Les encodages correspondant à $e = 1024$, ce qui équivaut à $e' = 2047 = (11111111111)_2$, sont utilisés pour représenter deux valeurs particulières :

- l'infini, codé avec une mantisse nulle, qu'on obtient lorsqu'un calcul avec des nombres flottants dépasse le plus grand nombre représentable ;
- le NaN (Not a Number), codé avec une mantisse non nulle, qu'on obtient comme résultat d'une opération invalide.

Exercice 6

On utilise la norme IEEE-754 sur une architecture à 64 bits.

1. Déterminer le plus petit / grand nombre positif représentable sous forme normalisée.
2. Déterminer le plus petit / grand nombre strictement positif représentable sous forme dénormalisée.

Remarques 7

a) Si un nombre dyadique x admet plus de 52 bits dans sa mantisse, alors l'ordinateur arrondira le nombre x pour ne conserver que 52 bits.

b) L'interpréteur permet de saisir des nombres décimaux, comme sur l'exemple ci-dessous.

```
>>> x = 0.1
```

Remarques 7

b) En général, un nombre décimal x n'est pas un nombre dyadique : il n'est donc pas possible de l'écrire avec une mantisse admettant un nombre fini de bits. Par exemple, on a

$$0,1 = 2^{-4} \times \left(1,1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ldots\right)_2.$$

Dans ce cas, l'ordinateur se contentera d'approcher $0,1$ par un nombre dyadique en arrondissant au plus proche. Par exemple, le nombre décimal $0,1$ est représenté en mémoire par

$$0{,}1 \approx 2^{-4} \times \Big(1,\underbrace{1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1010}_{52\text{bits}}\Big)_2$$

c) Dans les deux situations décrites ci-dessus, le codage de x sur 64 bits représentera une valeur qui n'est pas exactement égale à x, mais qui en sera très proche.

Exercice 7

Coder le nombre $125,75$ avec la norme IEEE-754 sur 64 bits.

Exercice 8

Décoder le nombre représenté avec la norme IEEE-754 sur 64 bits par

$$1 \quad 10000001100 \quad 1101010101010101000000000000000000000000000000000000.$$

Nous venons de voir que dans certains cas, l'ordinateur effectue des approximations afin de pouvoir coder les nombres en mémoire. Dans cette partie, nous allons également voir que les opérations usuelles sur les nombres flottants peuvent conduire à d'autres approximations.

Pour illustrer le cas de l'addition de deux flottants, nous utiliserons les deux nombres ci-dessous avec leur représentation en mémoire.

$$0{,}1 \approx 2^{-4} \times \left(1{,}1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1010\right)_2$$

$$0{,}2 \approx 2^{-3} \times \left(1{,}1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1010\right)_2$$

Pour additionner (ou soustraire) ces deux flottants, l'ordinateur commence par aligner les deux nombres pour qu'ils aient le même exposant (en prenant comme exposant commun le plus grand des deux).

$$0{,}1 \approx 2^{-3} \times \left(0{,}1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1100\ 1101\right)_2$$

$$0{,}2 \approx 2^{-3} \times \left(1{,}1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1001\ 1010\right)_2$$

L'ordinateur additionne les deux représentations.

$$0{,}1+0{,}2 \approx 2^{-3} \times \left(10{,}0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0110\ 0111\right)_2$$

Finalement, l'ordinateur normalise le résultat ci-dessus en arrondissant la mantisse au plus proche pour ne conserver que 52 bits.

$$0{,}1+0{,}2 \approx 2^{-2} \times \left(1{,}0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0011\ 0100\right)_2$$

Exemple 12

En particulier, en Python, la somme de $0,1$ et de $0,2$ n'est pas exactement égale à $0,3$. L'ordinateur effectue une approximation pour coder $0,1$ en mémoire, une approximation pour coder $0,2$ en mémoire et une approximation lors du calcul de leur somme.

```
>>> 0.1 + 0.2
0.30000000000000004
```

Une autre conséquence des approximations effectuées lors d'une addition avec des nombres flottants est le **phénomène d'absorption** qui se produit lorsqu'on additionne ou soustrait deux nombres flottants dont l'écart relatif est très important. Dans ce cas, la plus petite de ces quantités est absorbée par la plus grande.

```
>>> 2.**30 + 2.**(-23) == 2.**30
True
```

En fait, pour effectuer l'addition des nombres flottants 2^{30} et 2^{-23}, l'ordinateur va aligner leur représentation.

$$\begin{aligned} 2^{30} &= 2^{30} \times \left(1{,}0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\right)_2 \\ 2^{-23} &= 2^{30} \times \left(0{,}0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1\right)_2. \end{aligned}$$

L'ordinateur additionne les deux représentations ci-dessus, puis arrondit le résultat pour ne conserver que les 52 premiers bits de la mantisse.

$$\begin{aligned} 2^{30}+2^{-23} &= 2^{30} \times \left(1{,}0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1\right)_2 \\ &\approx 2^{30} \times \left(1{,}0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\right)_2 \\ &= 2^{30}, \end{aligned}$$

ce qui explique le phénomène d'absorption sur cet exemple.

Un autre problème peut se produire lorsqu'on effectue la différence de deux nombres flottants très proche : il s'agit du **phénomène d'annulation**. Dans cette situation, il est possible de perdre de nombreux chiffres significatifs, alors même que le résultat théorique peut être stocké dans un flottant.

```
>>> a = 2**(1/2) * (1 + 10**(-15))
>>> b = 2**(1/2)
>>> a - b
1.5543122344752192e-15
```

Le résultat obtenu est assez éloigné de celui attendu : la différence des deux quantités `a` et `b` qui sont très proches a annulé de nombreux chiffres significatifs.

Remarque 8

Le produit de deux nombres flottants est simple à effectuer :

1) on utilise la règle des signes pour déterminer le signe du produit ;
2) on additionne les exposants ;
3) on multiplie les mantisses ;
4) on normalise si nécessaire le résultat obtenu.

De la même manière que pour l'addition, il est parfois nécessaire d'arrondir le résultat final pour ne conserver que 52 bits dans la mantisse.

Bilan

Tout calcul sur les nombres flottants est accompagné d'une marge d'erreur. En particulier, on ne teste jamais l'égalité entre deux nombres flottants avec le symbole `==` : il est préférable de vérifier si l'écart entre eux est inférieur à une marge d'erreur choisie.

```
>>> 0.1 + 0.2 == 0.3
False
>>> marge_erreur = 10**(-15)
>>> abs(0.3 - (0.1 + 0.2)) < marge_erreur
True
```

Dans l'exemple ci-dessus, deux nombres flottants dont l'écart est inférieur à `marge_erreur` sont considérés comme égaux.