Chapitre (S2) 6 Représentation des nombres en machine

Objectifs

- Appréhender les limitations intrinsèques à la représentation et la manipulation informatique des nombres.
- Initier un sens critique au sujet de la qualité et de la précision des résultats de calculs numériques sur ordinateur.

1. REPRÉSENTATION DES ENTIERS

1.1. Entiers positifs

ÉCRITURE DÉCIMALE & BINAIRE Les entiers naturels sont usuellement exprimés sous la forme de leur *écriture décimale*, chaque chiffre (choisis parmi les dix entiers de 0 à 9) de la droite vers la gauche indiquant le nombre correspondant de « paquets » des puissances de 10 successives.

Exemple 1 Par exemple l'écriture décimale 12345 désigne le résultat du calcul :

$$1\times\underbrace{10000}_{10^4}+2\times\underbrace{1000}_{10^3}+3\times\underbrace{100}_{10^2}+4\times\underbrace{10}_{10^1}+5\times\underbrace{1}_{10^0}.$$

Sur le même modèle, il est bien sûr possible d'écrire les entiers naturels dans d'autres bases que la base 10. En particulier le choix de la base 2 conduit à l'*écriture binaire*, pour laquelle les chiffres ne peuvent prendre que les valeurs 0 ou 1, et indiquent un nombre de « paquets » de 2.

Exemple 2 Par exemple l'écriture binaire 101001 désigne le résultat du calcul

$$1\times\underbrace{32}_{2^5}+0\times\underbrace{16}_{2^4}+1\times\underbrace{8}_{2^3}+0\times\underbrace{4}_{2^2}+0\times\underbrace{2}_{2^1}+1\times\underbrace{1}_{2^0}.$$

Cet entier correspond donc à l'écriture décimale 41. Notons que, hors contexte, l'écriture 101001 est ambiguë car elle pourrait tout aussi bien désigner une écriture décimale (l'entier « cent-un-mille-un »). Lorsqu'il est nécessaire de lever cette source de confusion, on peut indiquer la base utilisée de la manière suivante : $(101001)_2$ désigne l'écriture binaire et $(101001)_{10}$ l'écriture décimale. De manière générale, nous avons le théorème suivant.

Théorème 1 | Existence et unicité de l'écriture en base b
Soient $b\geq 2$ et $n\geqslant 1$ deux entiers. Alors :

$$\exists!\,\mathrm{N}\geqslant 1,\ \exists!\,(c_0,\dots,c_{p-1})\in[\![0\,,\,b-1]\!]^p,\quad n=\sum_{k=0}^{\mathrm{N}-1}c_k b^k\ \ \underline{\textbf{et}}\ \ c_{\mathrm{N}-1}\neq 0.$$

Preuve Récurrence forte sur n.

- Comme nous l'avons vu sur l'exemple précédent, la représentation décimale d'un entier donné par son écriture binaire $(c_{\mathrm{N}-1}c_{\mathrm{N}-2}\dots c_1c_0)_2$ se fait en calculant la somme : $\sum_{k=0}^{\mathrm{N}-1}c_k\times 2^k$.
- À l'inverse, on peut déterminer les chiffres de la représentation binaire d'un entier connu par sa représentation décimale en calculant les restes successifs de sa division euclidienne par 2, jusqu'à ce que le quotient soit égal à 0. En effet,
  - ◇ $n=c_0+b\sum_{k=1}^{\mathrm{N}-1}c_k b^{k-1}$, donc : $c_0=n\%b$.
  - ◇ Ensuite, comme $n//b=\sum_{k=1}^{\mathrm{N}-1}c_k b^{k-1}$ on obtient comme précédemment en sortant le 1er terme : $c_1=(n//b)\%b$.
  - ◇ Et ainsi de suite....

Exemple 3

Par exemple, l'écriture binaire de l'entier $(25)_{10}$ peut s'obtenir après les calculs suivants :

25	2				
1	12	2			
	0	6	2		
		0	3	2	
			1	1	2
				1	0

ce qui donne la représentation binaire $(11001)_2$ (encore une fois, attention à l'ordre !).

Attention ! les chiffres obtenus sont à placer **de droite à gauche** dans l'écriture binaire (le chiffre de gauche étant appelé **bit de poids fort**).

Notons qu'un simple test de parité permet de savoir si le reste vaut 0 ou 1.

Remarque 1 D'autre part, on pourra préférer la présentation simplifiée suivante des divisions euclidiennes par 2, en indiquant sur deux lignes, **de la droite vers la gauche**, les quotients et restes des divisions euclidiennes successives par 2. Les différentes étapes de la construction de ce tableau pour le même exemple sont reproduites ci-dessous :

					25

				12	25
					1

			6	12	25
				0	1

		3	6	12	25
			0	0	1

	1	3	6	12	25
		1	0	0	1

0	1	3	6	12	25
	1	1	0	0	1

La deuxième ligne donne alors l'écriture binaire (**dans le bon sens...**)

Opérations. Rappelons le principe des retenues vu à l'école primaire, par exemple pour l'addition. Supposons que l'on souhaite additionner $n = a_1 \times 10 + a_0$ et $m = b_1 \times 10 + b_0$, avec $(a_1, a_0, b_1, b_0) \in [\![0\,,9]\!]^4$. Alors : $n + m = (a_1 + b_1) \times 10 + (a_0 + b_0)$. Plusieurs cas se présentent alors :

- si $0 \leqslant a_1 + b_1 < 10$ et $0 \leqslant a_0 + b_0 < 10$, alors $n + m = (a_1 + b_1) \times 10 + (a_0 + b_0)$ est bien la décomposition en base 10 de $n + m$.
- Si par exemple $a_0 + b_0 \geqslant 10$, alors on doit « transférer un paquet de 10 » des unités vers la gauche. On écrit alors :
  $$n + m = (a_1 + b_1 + 1) \times 10 + (a_0 + b_0 - 10).$$
  ◇ Si $a_1 + b_1 + 1 < 10$, c'est terminé.
  ◇ Sinon, on recommence comme avant en écrivant :
  $$n + m = 1 \times 10^2 + (a_1 + b_1 + 1 - 10) \times 10 + (a_0 + b_0 - 10).$$
  C'est l'étape de « transfert de 10 » qui se traduit par le système de retenue lorsque l'on pose l'opération.

Les opérations arithmétiques d'addition et multiplication s'effectuent avec les représentations binaires en utilisant le même principe qu'en base 10. Par exemple, posons en binaire le calcul de la somme de 22 et 15, dont les écritures binaires sont respectivement 10110 et 1111 :

$$\begin{array}{r} {\scriptstyle 1\ \ 1\ \ 1\ \ 1\ \ \ } \\ 1\,0\,1\,1\,0 \\ +\quad 1\,1\,1\,1 \\ \hline =1\,0\,0\,1\,0\,1 \end{array}$$

Vous pourrez vérifier que le résultat obtenu est bien l'écriture binaire de $22+15=37$. (Notez l'utilisation de retenues placées entre parenthèses au-dessus de la première ligne.)

Exercice 1 [Sol 1] Calculer la somme $13 + 11$ et le produit 13×11 en utilisant les représentations binaires de ces nombres.

Représentation des entiers positifs sur N bits Le stockage des entiers positifs dans la mémoire des ordinateurs se fait pour la plupart des langages de programmation en mémorisant les chiffres de son écriture binaire sur un nombre fixé de bits (en général sur 4 ou 8 octets, c'est-à-dire sur 32 ou 64 bits, suivant les usages). Nous verrons néanmoins à la fin de cette section que le langage Python constitue sur ce point une exception.

Bien sûr, des 0 sont ajoutés si nécessaire à gauche des chiffres de l'écriture binaire pour atteindre le nombre souhaité de bits. Par exemple, nous avons vu que l'entier 25 s'écrit en binaire $(11001)_2$ et sera stocké sur 8 bits sous la forme $(00011001)_2$.

En consacrant un nombre N de bits fixé à l'avance pour le stockage des chiffres binaires $(c_{N-1}c_{N-2}\dots c_1c_0)_2$ d'un entier positif, on limite la plage des entiers représentables. Par exemple, l'entier 25 s'écrit avec au moins cinq chiffres binaire, et ne pourrait pas par conséquent être stocké sur quatre bits seulement.

Plus précisément, le choix de stocker les entiers positifs sur N bits permet de représenter uniquement les entiers entre :

- $0 = (00\dots00)_2$,
- et : $\displaystyle\sum_{k=0}^{N-1} 1 \times 2^k = 2^N - 1 = (11\dots11)_2$.
- En résumé, les entiers de l'intervalle : $[\![0\,,2^N-1]\!]$.

Comme on pouvait s'y attendre (c'est le nombre de N-listes dans un ensemble à deux éléments), il y a donc 2^N entiers représentables sur N bits.

Exemple 4 (Différentes plages d'entiers codés sur N bits)
- Avec un stockage sur N = 4 bits on représente les entiers de l'intervalle $[\![0, 15]\!]$ uniquement.
- Cette plage s'accroît très largement sur N = 32 bits en passant à $[\![0\,,\,4294967295]\!]$,
- et encore plus sur N = 64 bits puisqu'elle devient $[\![0, 18446744073709551615]\!]$ (ce qui paraît assez largement suffisant pour les usages pratiques).

Néanmoins, une conséquence importante de la limitation du nombre de bits accordé au stockage est que les calculs arithmétiques effectués dans ce cadre ne sont plus nécessairement exacts. Illustrons ce phénomène en revenant sur l'exemple du calcul en binaire de 22 + 15, mais en limitant le stockage à N = 5 bits (qui permet de stocker les entiers de 0 à 31, donc 22 et 15 sont bien tous les deux représentables) :

$$\begin{array}{r} {\scriptstyle 1\ \ 1\ \ 1\ \ \ \ \ } \\ 1\,0\,1\,1\,0 \\ +\,0\,1\,1\,1\,1 \\ \hline =\,0\,0\,1\,0\,1 \end{array}$$

La différence avec le calcul précédent, qui ne limitait pas le nombre de chiffres binaires, est que l'on ne peut plus reporter la dernière retenue. Cette opération ainsi tronquée conduit ainsi à l'horrible égalité : « 22 + 15 = 5 ».

En fait, la disparition de la dernière retenue, qui correspond à un « paquet » de 2^N, permet de constater que si un calcul restreint à N bits n'est pas nécessairement correct, il l'est malgré tout toujours au moins **modulo** 2^N. Il faut donc plutôt lire ici :

$$22 + 15 \equiv 5 \quad [32]\,.$$

CODAGE DES FONCTIONS DE CONVERSION Dans ce paragraphe, on chercher à coder les différentes fonctions Python permettant de calculer la décomposition d'un entier dans une certaine base, ainsi que l'opérations réciproque.

Exercice 2 Entier vers écriture décimale [Sol 2] Écrire une fonction `dec2bin(n : int, N : int) -> list` renvoyant la liste formant l'écriture binaire sur N bits de l'entier positif d'écriture décimale n.

Par exemple, `dec2bin(25,6)` renverra la liste `[0, 1, 1, 0, 0, 1]`.

L'entier positif n est supposé être représentable sur N bits, ce dont la fonction s'assurera en utilisant une instruction `assert`.

Exercice 3 Écriture décimale vers entier [Sol 3] On souhaite écrire une fonction `bin2dec(L : list) -> int` renvoyant l'écriture décimale de l'entier positif dont la liste L contient l'écriture binaire.
Par exemple, `bin2dec([0, 1, 1, 0, 0, 1])` renverra l'entier 25.

1) Proposer une première version de cette fonction qui, pour $L = [c_{N-1}\,c_{N-2}\ldots c_1\,c_0]$, calcule la somme $\sum_{k=0}^{N-1} c_k \times 2^k$ sans chercher à optimiser l'évaluation des puissances de 2. Exprimer, en fonction de N =`len(L)`, la complexité de cette fonction (le calcul de 2^k comptera pour k opérations).

2) On souhaite maintenant réduire cette complexité en optimisant le calcul des puissances.

2.1) On note $L' = [c_{N-1}\,c_{N-2}\ldots c_1]$ la sous liste de L amputée de son dernier élément. Déterminer une relation liant l'entier n représenté par L et l'entier n' représenté par L′.

2.2) En déduire une version **récursive** de la fonction `bin2dec`.

2.3) Préciser la complexité de cette version.

2.4) Écrire une version itérative de la fonction `bin2dec` fondée sur la même idée.

1.2. Entiers signés

Dans cette partie, on s'intéresse à la représentation binaire sur N bits d'un entier **relatif**.

UNE PREMIÈRE (MAUVAISE) IDÉE Pour représenter un entier relatif n, une idée assez naturelle consiste à réserver un premier bit pour le signe (par exemple 0 si n est positif et 1 s'il est négatif) et stocker la valeur absolue de n sur les N − 1 bits restants.

Par exemple, sur 8 bits :

- 3 serait représenté par 00000011 ;
- −4 serait représenté par 10000100.

Mais le problème majeur de cette représentation (outre le fait que 0 serait codé de deux manières différentes : $000\ldots0$ et $100\ldots0$) est que les algorithmes usuels pour les opérations arithmétiques ne fonctionnent plus. Par exemple, le calcul de $3+(-4)$, toujours sur 8 bits, donne :

$$\begin{array}{r} 0\,0\,0\,0\,0\,0\,1\,1 \\ +\,1\,0\,0\,0\,0\,1\,0\,0 \\ \hline 1\,0\,0\,0\,0\,1\,1\,1 \end{array}$$

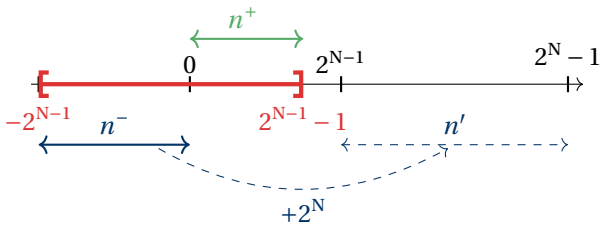
Le résultat, correspondant à l'entier relatif -7, est bien sûr faux (même modulo $2^8 = 256$).

UNE MEILLEURE SOLUTION Nous allons donc nous utiliser une représentation des entiers relatifs plus sophistiquée, mais qui présentera l'avantage de pouvoir conserver les algorithmes usuels des opérations arithmétiques vus pour les entiers positifs, et qui possède toujours une plage d'entiers représentables égale à $[\![0\,,\,2^N-1]\!]$.

L'idée est de « couper en deux » cet intervalle, dédier la moitié aux positifs et l'autre moitié aux négatifs. *(en valeur absolue, on va donc pouvoir représenter moins d'entiers que dans le cas des naturels)* Plus précisément, on va représenter sur N bits les entiers n de l'intervalle $[\![-2^{N-1}\,,\,2^{N-1}-1]\!]$ (noter qu'il y aura donc un entier négatif de plus que les entiers positifs, et qu'il y aura $(2^{N-1}-1)-(-2^{N-1})+1 = 2^N$ entiers relatifs codés, ce qui coïncide avec la représentation des entiers naturels sur N bits vue précédemment) de la manière ci-après.

Principe de codage des entiers relatifs $n \in \mathbb{Z}$ sur N bits

- **[Cas $n \geqslant 0$]** c'est-à-dire $n \in [\![0\,,\,2^{N-1}-1]\!]$, alors n est codé sur N bits en tant qu'entier naturel, comme nous l'avons vu dans la partie précédente ;
- **[Cas $n < 0$]** c'est-à-dire $n \in [\![-2^{N-1}\,,\,-1]\!]$, alors on calcule le nouvel entier décalé $n' = n + 2^N$ (donc $n' \in [\![2^{N-1}\,,\,2^N-1]\!]$) dont le codage sur N bits en tant qu'entier naturel sera la représentation de n.



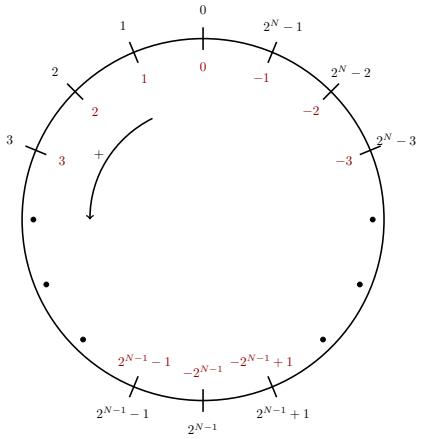
(n^+ codé de manière habituelle, n^- après décalage)

FIGURE 1. – Codage des entiers signés

Nombre	Représentation
0	0000……0000
1	0000……0001
…	……
$2^{n-1}-1$	0111……1111
-2^{n-1}	1000……0000
…	……
-1	1111……1111

FIGURE 2. – Les entiers relatifs codés sur n bits, classés par représentation croissante.

Ce principe est tout compte fait très naturel, si on comprend que dans tous les cas on souhaite coder sur N bits un entier naturel appartenant à l'intervalle $[\![0\,,\,2^N-1]\!]$, soit directement l'entier n s'il est positif, soit l'entier n' congru à n modulo 2^N qui appartient à cet intervalle. Le diagramme circulaire suivant (les abscisses se répétant modulo 2^N) illustre la situation, l'entier relatif n à coder étant indiqué à l'intérieur du cercle et l'entier naturel codé réellement indiqué à l'extérieur (c'est n pour la partie gauche du cercle, et $n' = n + 2^N$ pour la partie droite).



Exemple 5 Par exemple, sur N = 8 bits, on code les entiers relatifs de l'intervalle $[\![-2^7, 2^7-1]\!]$, ie. $[\![-128, 127]\!]$.

- L'entier relatif $n = 45$ se code 00101101 (puisque n est positif, il se code directement comme entier naturel sur 8 bits).
- L'entier relatif $n = -45$ se code 11010011 (puisque n est strictement négatif, on code l'entier naturel $n' = n + 2^8 = -45 + 256 = 211$ sur 8 bits).

Attention

Attention donc à bien distinguer un nombre de sa représentation en machine, qui ne coïncide pas avec sa représentation binaire dans le cas des nombres négatifs.

Notons qu'avec cette représentation, le bit de poids fort (celui de gauche) de l'écriture binaire vaut toujours 0 si l'entier relatif n est positif, et vaut 1 s'il est strictement négatif. En effet, on vérifie aisément que le bit de poids fort de l'écriture binaire d'un entier naturel vaut 0 s'il appartient à l'intervalle $[\![0, 2^{N-1}-1]\!]$, et vaut 1 s'il appartient à l'intervalle $[\![2^{N-1}, 2^N-1]\!]$.

Remarque 2 Ainsi, comme pour la premier idée naïve de codage, le premier bit est un **bit de signe**, mais la différence est que dans le cas où l'entier est strictement négatif les bits suivants ne codent pas sa valeur absolue !

Principe de décodage des entiers relatifs $n \in \mathbb{Z}$ sur N bits

Le décodage d'une écriture binaire pour obtenir l'entier relatif n correspondant peut alors se faire de la manière suivante :

- si le bit de poids fort vaut 0 : on décode la succession de bits comme un entier naturel, et cet entier est n ;
- si le bit de poids fort vaut 1 : on décode la succession de bits comme un entier naturel n', et alors $n = n' - 2^N$.

Exemple 6 Par exemple, toujours sur N = 8 bits :

- 00010110 a un poids fort égal à 0, donc le codage correspond à l'entier relatif $n = 22$.
- 10010110 a un poids fort égal à 1, et le codage correspond à l'entier naturel $n' = 150$, donc à l'entier relatif $n = n' - 2^8 = 150 - 256 = -106$.

Enfin, notons que l'entier relatif représenté est toujours congru à l'entier naturel effectivement codé modulo 2^N (soit ils sont égaux, soit la différence vaut 2^N), et comme on sait que les opérations arithmétiques d'addition et multiplication sont compatibles avec la congruence, on est assuré que les ces opérations sur les entiers relatifs peuvent s'effectuer directement sur les entiers entiers naturels qui les codent. Le résultat donnera bien un entier relatif soit égal au résultat attendu, ou au pire congru modulo 2^N.

Exercice 4 [Sol 4] Effectuer les calculs $3 + (-4)$ et $3 \times (-4)$ à partir de leurs représentations binaires sur 8 bits.

CODAGE PAR COMPLÉMENT À DEUX Pour coder sur N bits un entier $n \in [\![-2^{N-1}, 2^{N-1}-1]\!]$, dans le cas où n est strictement négatif, nous avons vu qu'il fallait coder l'entier naturel $n' = n + 2^N$. Or il existe une méthode, dite *méthode du complément à deux*, qui dispense d'effectuer cette addition (que l'ordinateur d'ailleurs ne saurait pas faire en pratique, puisqu'il ne saura ajouter des entiers qu'une fois qu'ils seront représentés en binaire).

Principe du complément à 2 (codage de $n' = n + 2^N$, $n < 0$)

- On code l'entier naturel $-n$ sur les N bits ;
- puis on inverse tous les bits de cette écriture (les 0 en 1, et inversement) ;
- et pour finir, on ajoute 1 au résultat (addition binaire sur N bits, **sans propager au-delà une éventuelle dernière retenue**).

Exemple 7 Par exemple, reprenons par cette méthode le codage de $n = -45$ sur 8 bits :

- on code $-n = 45$, ce qui donne 00101101 ;
- on inverse tous les bits, ce qui donne 11010010 ;
- on ajoute 1 au résultat, ce qui donne 11010011.

Preuve *(Complément à 2 (codage))* Notons $c_{N-1}c_{N-2}\dots c_1c_0$ le codage de $-n$ (on a donc $-n=\sum_{k=0}^{N-1}c_k2^k$). En inversant tous les bits on écrit $d_{N-1}d_{N-2}\dots d_1d_0$ avec $d_k=1-c_k$, et en ajoutant 1 on obtient le codage de l'entier

$$1+\sum_{k=0}^{N-1}d_k2^k=1+\sum_{k=0}^{N-1}(1-c_k)2^k=1+\underbrace{\sum_{k=0}^{N-1}2^k}_{\frac{1-2^N}{1-2}=2^N-1}-\underbrace{\sum_{k=0}^{N-1}c_k2^k}_{-n}=n+2^N=n'.$$

Principe du décodage du complément à 2 (décodage de $n'=n+2^N$, $n<0$)

À l'inverse, on peut retrouver l'entier relatif correspondant à une écriture binaire, dans le cas où le bit de poids fort vaut 1, de la manière suivante :

- on inverse tous les bits de cette écriture ;
- puis on ajoute 1 au résultat (addition binaire sur N bits, **sans propager au-delà une éventuelle dernière retenue**) ;
- on décode l'entier naturel obtenu : l'opposé de ce nombre est le résultat.

Exemple 8 Par exemple, reprenons par cette méthode le décodage de 10010110 (dont le bit de poids fort est bien égal à 1) :

- on inverse tous les bits, ce qui donne 01101001 ;
- on ajoute 1, ce qui donne 01101010 ;
- on décode l'entier naturel 106, donc le résultat est -106.

Preuve *(Complément à 2 (décodage))* Notons n l'entier relatif de codage $c_{N-1}c_{N-2}\dots c_1c_0$ avec $c_{N-1}=1$ (donc $2^N+n=n'=\sum_{k=0}^{N-1}c_k2^k$). En inversant tous les bits on écrit $d_{N-1}d_{N-2}\dots d_1d_0$ avec $d_k=1-c_k$, et en ajoutant 1 on obtient le codage de l'entier

$$1+\sum_{k=0}^{N-1}d_k2^k=1+\sum_{k=0}^{N-1}(1-c_k)2^k=1+\underbrace{\sum_{k=0}^{N-1}2^k}_{\frac{1-2^N}{1-2}=2^N-1}-\underbrace{\sum_{k=0}^{N-1}c_k2^k}_{2^N+n}=-n.$$

CODAGE DES FONCTIONS DE CONVERSION. Comme pour les entiers naturels, on s'intéresse à présent au codage des entiers signés à l'aide de Python.

Exercice 5 [Sol 5] Cet exercice utilise les fonctions `dec2bin` et `bin2dec` écrites dans les exercices 1.1 et 1.1.

1) Écrire une fonction `dec2binSigne(n : int, N : int) -> list` renvoyant la liste formant l'écriture binaire sur N bits de l'entier signé d'écriture décimale n.
Par exemple, `dec2binSigne(-45,8)` renverra la liste `[1, 1, 0, 1, 0, 0, 1, 1]`.
L'entier signé n est supposé être représentable sur N bits, ce dont la fonction s'assurera en utilisant une instruction `assert`.
La méthode utilisée sera celle utilisant une addition pour calculer l'entier n' si nécessaire (la méthode utilisant le complément à deux sera programmée en TP).

2) Écrire une fonction `bin2decSigne(L : list) -> int` renvoyant l'écriture décimale de l'entier signé dont la liste L (non vide) contient l'écriture binaire.
Par exemple, `bin2decSigne([1, 0, 0, 1, 0, 1, 1, 0])` renverra l'entier `-106`.
Dans cette question également, la méthode utilisée ne sera pas celle du complément à deux.

1.3. Et en langage Python ?

Dans de nombreux langages, les entiers signés sont codés avec un nombre de bits N fixé une fois pour toutes, et dépendant du type choisi pour la variable. On peut avoir, par exemple, pour le type `int`, un codage sur N = 32 bits « uniquement », ce qui correspond à un intervalle représenté $[\![-2^{31};2^{31}-1]\!]$, soit $[\![-214748348;214748347]\!]$ (c'était le cas pour le langage Python jusqu'aux versions 2.xx). Les opérations sur ces entiers peuvent alors, comme nous l'avons vu, entraîner des dépassements de capacité qui conduisent à des résultats erronés (exacts seulement modulo 2^{32}).

Depuis les versions 3.xx, pour éviter ce problème en cas de dépassement, le langage Python modifie *dynamiquement* la taille mémoire allouée au stockage des entiers manipulés pour permettre une représentation et des calculs **exacts**. Le type `int` dans le langage Python n'est donc pas de taille prédéfinie, mais permet de représenter des entiers arbitrairement longs (leurs tailles n'est en pratique limitée que par la quantité de mémoire disponible sur la machine). Notons qu'une difficulté apparaît alors pour les calculs de complexité, puisque le temps de calcul correspondant à un même type d'opération sur des entiers (par exemple une addition) peut fortement dépendre de la taille effective de leurs représentations, ce que nous négligeons habituellement en pratique.

2. REPRÉSENTATION DES RÉELS

2.1. Notation scientifique décimale (ou représentation à virgule flottante)

Revenons temporairement à la notation décimale, mais cette fois-ci pour les réels. Pour les nombres réels, il est fréquent d'utiliser la notation scientifique, comme dans

l'exemple ci dessous :

- $\pi = 3,14159265$
- $c = 2,99792458 \times 10^{8}$,
- $N_A = 6,022142 \times 10^{23}$
- $h = 6,62606957 \times 10^{-34}$

Théorème 2 | Écriture en notation scientifique

On peut écrire tout réel x sous la forme suivante :

$$x = (-1)^s \times 10^{E} \times (d_0 + d_1 \times 10^{-1} + d_2 \times 10^{-2} + \cdots + d_i \times 10^{-i} + \cdots)$$
$$= (-1)^s \times 10^{E} \times \sum_{i=0}^{\infty} 10^{-i} d_i.$$

avec :

- $s = 0$ ou $s = 1$ (fixe le signe de x).
- E est un entier relatif.
- $d_i \in \{0, 1, 2, \cdots, 9\}$, avec $d_0 \neq 0$.

Remarque 3 (Aspects mathématiques)

- On peut montrer que $d_i = \lfloor 10^i x \rfloor - 10 \times \lfloor 10^{i-1} x \rfloor$ convient, et :

$$\text{E tel que :} \quad 1 \leqslant \frac{|x|}{10^{E}} < 10 \iff E = \lfloor \log |x| \rfloor.$$

- La convergence de la série est garantie par le théorème de comparaison pour les séries à termes positifs, et combinée avec la majoration :

$$\forall i \in \mathbb{N}, \quad 10^{-i} d_i \leqslant \underbrace{9 \times \left(\frac{1}{10}\right)^{i}}_{\text{t.g. d'une série convergente}}.$$

Dans cette décomposition, le nombre de décimales peut être infini *(on a une série)*, par exemple $\frac{4}{3} = 1,33333\cdots$, ou pas : $\frac{5}{4} = 1,25$.

2.2. Les réels en base binaire

On étend sans difficulté l'écriture de nombres à virgules en binaire. Par exemple, $(110,01)_2$ désignera le flottant :

$$1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 + 0 \times \frac{1}{2^1} + 1 \times \frac{1}{2^2}$$
$$= 2^2 \left(1 + 1 \times 2^1 + 0 \times 2^0 + 0 \times \frac{1}{2^1} + 1 \times \frac{1}{2^2}\right). \quad (\star)$$

Dans la base binaire, on a le même genre de résultats que pour la notation scientifique, à certaines transpositions près : c'est exactement l'écriture $(\star)$.

Théorème 3 | Écriture « normalisée »

On peut écrire tout réel x sous la forme suivante :

$$x = (-1)^s \times 2^{E} \times \left(b_0 + b_1 \times 2^{-1} + b_2 \times 2^{-2} + \cdots + b_i \times 2^{-i} + \cdots\right)$$
$$= (-1)^s \times 2^{E} \times \sum_{i=0}^{\infty} 2^{-i} b_i.$$

avec :

- $s = 0$ ou $s = 1$ (fixe le signe de x).
- E est un entier relatif.
- $b_i \in \{0, 1\}$, avec $b_0 \neq 0$ (c'est-à-dire $b_0 = 1$).

Remarque 4 (Aspects mathématiques)

- On peut montrer que $b_i = \lfloor 2^i x \rfloor - 2 \times \lfloor 2^{i-1} x \rfloor$ convient, et :

$$\text{E tel que :} \quad 1 \leqslant \frac{|x|}{2^{E}} < 2 \iff E = \lfloor \log_2 |x| \rfloor.$$

- La convergence de la série se justifie comme précédemment.

Pour déterminer x, il faut connaître S, E et $(b_i)_{i \in \mathbb{N}}$.

2.3. Écriture normalisée & Norme IEEE-754

La taille de la mémoire machine étant limitée, on a forcément l'exposant E qui appartient à un intervalle borné, et les décimales binaires $(b_i)_{i \in \mathbb{N}}$ qui sont en nombre fini.

Mantisse (tronquée) & son codage. Rappelons que nous avons montré que tout réel $x \in \mathbb{R}$ s'écrit de la manière suivante :

$$x = (-1)^s \times 2^{E} \times \left(1 + b_1 \times 2^{-1} + b_2 \times 2^{-2} + \cdots + b_i \times 2^{-i} + \cdots\right).$$

La « mantisse de x » en Mathématiques est le réel $1 + b_1 \times 2^{-1} + b_2 \times 2^{-2} + \cdots + b_i \times 2^{-i} + \cdots$. En informatique, on choisira de tronquer la somme $1 + b_1 \times 2^{-1} + b_2 \times 2^{-2} + \cdots + b_i \times 2^{-i} + \cdots$ en ne gardant que les premiers bits. On note m le nombre de bits utilisés pour coder cette somme.

Définition 1 | Mantisse

Soit $x = (-1)^s \times 2^{E} \times \left(b_0 + b_1 \times 2^{-1} + b_2 \times 2^{-2} + \cdots + b_i \times 2^{-i} + \cdots\right) \in \mathbb{R}$. On appelle *mantisse de x sur m bits* ou simplement *mantisse de x* la quantité :

$$M = 1 + b_1 \times 2^{-1} + b_2 \times 2^{-2} + \cdots + b_m \times 2^{-m}.$$

Remarque 5 Par majorations évidentes, on montre : $1 \leqslant M < 2$.

Le codage binaire de la mantisse se ramène à la donnée des $(b_i)_{1\leqslant i\leqslant m-1}$. On notera la mantisse simplement de la manière suivante :

$$b_1\, b_2 \dots b_m.$$

On remarque que dans cette écriture, le bit de poids le plus fort est placé à gauche, comme pour la représentation des entiers.

EXPOSANT & SON CODAGE. L'exposant E est un élément de $\mathbb{Z}$, on pourrait donc utiliser la section précédente pour le représenter. Cependant, une autre convention est utilisée par la machine ; celle de représenter les entiers relatifs de $[\![-2^{e-1}+1; 2^{e-1}]\!]$ (c'est toujours un ensemble ayant 2^e éléments) plutôt que $[\![-2^{e-1}, 2^{e-1}-1]\!]$ comme précédemment, où e est le nombre de bits utilisés. Comme ici nous n'avons aucun besoin de préserver les opérations arithmétiques sur les exposants, nous allons utiliser une solution plus naïve que pour le complément à deux ; celle de simplement décaler l'exposant pour le rendre positif. On code plus précisément en binaire l'entier naturel E′ défini par $\mathrm{E}' = \mathrm{E} + 2^{e-1} - 1$. On a alors $\mathrm{E}' \in [\![0; 2^e-1]\!]$, avec $\mathrm{E}' = \sum_{i=0}^{e-1} c_i 2^i$, que l'on notera :

$$c_{e-1} \dots c_1 c_0, \quad \text{avec } c_i \in \{0,1\}.$$

BILAN : REPRÉSENTATION BINAIRE D'UN RÉEL. La représentation d'un réel nécessite donc $m+e+1$ bits, et on peut proposer la représentation suivante :

$$\underbrace{s}_{\text{signe}}\ \underbrace{c_{e-1}\cdots c_1 c_0}_{\text{exposant E}'}\ \underbrace{b_1 b_2 \cdots b_m}_{\text{mantisse}}$$

où s est le bit de signe, les (c_j) fournissent la représentation binaire de $\mathrm{E}' = \mathrm{E} + 2^{e-1} - 1$ et les (b_i) fournissent la représentation binaire de la mantisse. On obtient alors la valeur de x par la formule : $x = (-1)^s \times 2^{\mathrm{E}} \times \mathrm{M}$, une fois les valeurs de $s, \mathrm{E}, \mathrm{M}$ récupérées (*via* des fonctions de décodage).

CAS PARTICULIER DU ZÉRO. Avec cette méthode de représentation, il est impossible de représenter la valeur 0, car $\mathrm{E} \geqslant -2^{e-1}+1$ et $\mathrm{M} \geqslant 1$, et donc pour tout réel x représenté, on a $|x| \geqslant 2^{-2^{e-1}+1}$. On choisit donc d'associer conventionnellement le codage du zéro aux valeurs ($\mathrm{E} = -2^{e-1}+1$, $b_i = 0$ pour tout i). Pour ce couple de valeurs, on a donc $\mathrm{E}' = 0$, $b_i = 0$ pour tout i. Le bit de signe n'est pas fixé. On a donc deux codages possibles pour ce zéro :

signe	exposant E'	mantisse
0	$0 \cdots 0$	$0 \cdots 0$
1	$0 \cdots 0$	$0 \cdots 0$

Remarque 6 (Convention : deux valeurs spéciales de E, nombres dénormalisés.) On distingue deux catégories de nombres particuliers, qui correspondent à deux valeurs particulières de l'exposant E.

- Le cas $\mathrm{E} = -2^{e-1}+1$ (ou $\mathrm{E}' = 0$), correspond au cas des nombres dits *dénormalisés*. Pour ces nombres, les machines utilisent une définition de mantisse M un peu différente, et devient $\mathrm{M} = b_1 \times 2^{-1} + b_2 \times 2^{-2} + \cdots + b_m \times 2^{-m}$. On peut donc avoir $\mathrm{M} \leqslant 1$. Cette optimisation permet une meilleure représentation des nombres au voisinage de 0 (mais nous ne rentrerons pas dans ces détails, très techniques, ces nombres de petite valeur absolue ne seront pas étudiés par la suite). On peut noter que le zéro réel est un cas particulier des nombres dénormalisés, pour lequel $\mathrm{E}' = 0$ et $\mathrm{M} = 0$.
- Le cas $\mathrm{E} = 2^{e-1}$ (ou $\mathrm{E}' = 2^e - 1$), est réservé en machine au codage de certains résultats non numériques, comme l'∞, ou comme `NaN` (Not a Number), qui peut être retourné en cas de calcul illicite, $\frac{0}{0}$ par exemple.

Résumé Représentations en base 2

Type d'objets	Nombre de bits	Entiers codés	Principe du codage
Entiers naturels n ($\star$)	N	$[\![0, 2^{\mathrm{N}}-1]\!]$	$(c_{\mathrm{N}-1} \dots c_0)$ avec $n = \sum_{k=0}^{\mathrm{N}-1} c_k 2^k$
Entiers relatifs n	N	$[\![-2^{\mathrm{N}-1}, 2^{\mathrm{N}-1}-1]\!]$	$n \geqslant 0$: codage ($\star$) de n, $n < 0$: codage ($\star$) de $n' = n + 2^{\mathrm{N}} - 1$
Nombres réels x	$x = (-1)^s \times 2^{\mathrm{E}} \times \mathrm{M}$, $\mathrm{M} \in [1,2[$		
	1	s	+ codé avec 0, − codé avec 1
	e	$\mathrm{E} \in [\![-2^{e-1}+1; 2^{e-1}]\!]$	$\mathrm{E}' = \mathrm{E} + 2^{e-1} - 1$ codage ($\star$)
	m	M	$1 + b_1 \times 2^{-1} + b_2 \times 2^{-2} + \cdots + b_i \times 2^{-i} + \cdots$, série tronquée sur m bits : $1 + b_1 \times 2^{-1} + b_2 \times 2^{-2} + \cdots + b_m \times 2^{-m}$

Cadre

Dans la suite, on se restreint aux nombres non dénormalisés et qui correspondent à des valeurs numériques, ce qui restreint l'intervalle des exposants E. On obtient donc finalement : $\mathrm{E}_{\text{restreint}} \in [\![-2^{e-1}+2; 2^{e-1}-1]\!]$.

2.4. Détermination pratique de la représentation

Dans l'écriture binaire, on a $x = (-1)^s 2^{\mathrm{E}} \mathrm{M}$, avec $1 \leqslant \mathrm{M} < 2$, nous avions déjà constaté que $\mathrm{E} = \lfloor \log_2 |x| \rfloor$, mais cette expression n'est pas satisfaisante d'un point de vue pratique. On utilise plutôt cet algorithme :

Calcul de E et M

Données : Un réel x

Résultat : La mantisse de x
$\mathrm{M} \leftarrow |x|$, $\mathrm{E} \leftarrow 0$.
- Tant que $\mathrm{M} < 1$ faire : $\mathrm{M} \leftarrow 2\mathrm{M}$ et $\mathrm{E} \leftarrow \mathrm{E} - 1$.
- Tant que $\mathrm{M} \geqslant 2$ faire : $\mathrm{M} \leftarrow \mathrm{M}/2$ et $\mathrm{E} \leftarrow \mathrm{E} + 1$.

renvoyer E, M.

De sorte qu'on a l'invariant : $|x| = 2^{\mathrm{E}}\mathrm{M}$. Il reste donc à déterminer la représentation binaire de M. Comme $1 \leqslant \mathrm{M} < 2$, on aura en binaire :

$$\mathrm{M} = (1, b_1 b_2 \cdots b_m)_2$$

Pour déterminer la suite des b_i, on utilise une suite y_i de premier terme $y_1 = \mathrm{M} - 1 = (0, b_1 b_2 \cdots b_m)_2$, et on a :

- si $y_i \geqslant 0.5$, alors $b_i = 1$ et $y_{i+1} = 2y_i - 1 = (0, b_{i+1} \cdots b_m)_2$;
- sinon $b_i = 0$ et $y_{i+1} = 2y_i = (0, b_{i+1} \cdots b_m)_2$.

On peut montrer (par récurrence sur i) que l'on obtient bien la suite des b_i correspondant à représentation binaire de M.

Calcul de la représentation binaire de la mantisse

Données : Un réel $\mathrm{M} \in [1, 2[$

Résultat : La liste des $(b_i)_{1 \leqslant i \leqslant m}$
$b = []$ et $y = \mathrm{M} - 1$
pour i allant de 1 à m faire :
- Si $y \geqslant 0.5$, $b \leftarrow b + [1]$, $y \leftarrow 2y - 1$.
- Sinon, $b \leftarrow b + [0]$, $y \leftarrow 2y$.

renvoyer b.

Exercice 6 [Sol 6]

1) Déterminer la représentation binaire $(s, \mathrm{E}, \mathrm{M})$ de $x = 1.0625$. On limitera la mantisse aux quatre premiers bits.

2) Même question avec $x = 0.1$, puis $x = -0.3$.

2.5. La norme IEEE754

Pour pouvoir échanger les données entre différents programmes et (ou) machines, il est nécessaire de fixer une norme commune, en particulier dans la représentation des nombres. La norme IEEE754 répond à cet objectif.

Dans cette norme, on adopte le principe précédemment décrit avec un codage sur 64 bits, répartis de la manière suivante :

- 1 bit pour le signe.
- 11 bits pour l'exposant. On a donc $e = 11$.
- 52 bits pour la mantisse. On a donc $m = 52$.

Dans la suite on s'intéresse à certaines propriétés de la représentation des nombres dans cette norme. On se limite pour l'instant aux seuls nombres normalisés.

2.6. Propriétés

Comme pour les entiers, l'ensemble des réels représentés est forcément fini, avec cependant une différence importante. Pour les entiers, on avait la possibilité de représenter la totalité des entiers contenus dans un certain intervalle. Pour les réels, ce n'est pas possible car tout intervalle contient une infinité de réels.

On représente donc les réels de manière approchée, et la taille finie de la mantisse impose un intervalle minimal entre deux réels représentés successifs.

Intervalle représenté On cherche à déterminer l'intervalle contenant tous les réels positifs dans cette norme, sous la forme $[x_m, x_M]$.

Exercice 7 [Sol 7]

1) Donner numériquement l'intervalle des exposants correspondant aux nombres normalisés.

2) Déterminer l'expression du plus grand réel positif représentable x_M, et donner sa valeur numérique sous la forme $r \times 10^n$ avec n entier et $|r| < 10$.

3) Déterminer l'expression du plus petit réel positif normalisé représentable x_m , et donner sa valeur numérique sous la forme $x \times 10^n$.

Précision absolue de la représentation On cherche à exprimer la différence entre deux nombres réels successifs x et x', notée $\delta x = |x' - x|$.

Exercice 8 [Sol 8]

1) Déterminer la plus petite variation de mantisse δM possible.

2) En déduire δx pour les plus petits nombres représentés, et pour les plus grands.

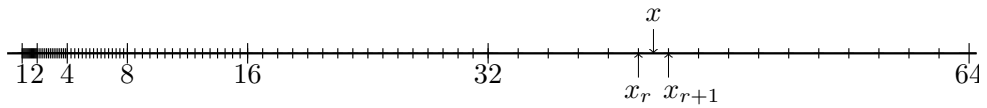
Précision relative de la représentation On peut s'intéresser à la précision relative entre ces deux mêmes nombres normalisés successifs. On définit la précision relative par : $\boxed{\varepsilon = \dfrac{|x' - x|}{|x|}}$.

Exercice 9 [Sol 9]

1) Déterminer les valeurs possibles de ε pour les nombres normalisés.

2) En déduire le nombre de chiffres significatifs correctement représentés pour les réels normalisés lorsqu'ils sont écrits sous la forme $r \times 10^n$.

Répartition des nombres représentés sur la droite réelle Si on place sur une droite graduée les réels représentables, alors dans tout intervalle $[2^p, 2^{p+1}[$ (avec $p \in [\![-2^{e-1}+2; 2^{e-1}-1]\!]$) on a des réels distants de 2^{p-m}. Pour p petit, deux nombres successifs sont plus proches que pour p grand. Dans le graphique ci-dessous, on a pris $m = 4$, ce qui fait $2^4 = 16$ réels représentables dans chaque intervalle de la forme $[2^p; 2^{p+1}[$, il y en a donc $\dfrac{2^4}{2^p}$ par unité de longueur, lorsque p augmente cette quantité tend vers 0, on a donc une raréfaction des nombres représentables lorsque ceux-ci augmentent. Lorsqu'un réel x est entre deux réels représentables consécutifs : $x \in [x_r; x_{r+1}[$, alors x est représenté par x_r ou x_{r+1} suivant la convention sur l'arrondi [1].



Pour un nombre total de bits fixé, on a $e + m$ constant. On a un compromis à trouver entre portée (intervalle des réels représentables) et précision. Pour augmenter la précision, il faut augmenter m, mais alors e diminue et on peut représenter moins de nombres. Pour augmenter la portée, il faut augmenter e, mais alors m diminue et la précision également.

2.7. Limitations

Cette représentation induit un certain nombre de comportements surprenants lorsqu'on effectue des calculs avec des réels, mais qui s'expliquent quand on tient compte de la manière dont ces réels sont représentés. Pour faciliter la prise en compte de ces défauts, on peut limiter ici la mantisse à quatre bits (soit $m = 4$). On notera x_r le nombre réel correspondant à la représentation binaire du nombre x, avec la convention de l'arrondi par valeur inférieure.

Somme de nombres de valeurs très différentes : phénomène d'absorption Effectuons la somme des réels $x = 1$ et $y = 0.03125$. La représentation binaire de ces nombres est $x_r = 2^0 \times 1$, $y_r = 2^{-5} \times 1$. Pour réaliser la somme, on écrit $y_r = 2^0 \times 2^{-5}$. On a alors en sommant $x_r + y_r = 2^0 \times (1 + 2^{-5})$. Mais le dernier bit n'est pas représentable avec $m = 4$. On tronque donc cette somme en ne gardant que quatre bits pour

1. Il y a plusieurs possibilités : arrondi par valeur inférieure (x_r), par valeur supérieure (x_{r+1}), ou bien au plus proche. La norme IEEE754 utilise l'arrondi au plus proche, mais pour simplifier l'exposé, dans la suite, les exemples seront traités avec l'arrondi inférieur.

la mantisse, et le résultat final est donc $2^0 \times 1$ pour $x_r + y_r$, soit la même valeur que x_r.

Plus généralement, on a :

$$2^e\left(1+\sum_{i=1}^{m}\frac{b_i}{2^i}\right)+2^{m+2+e}\left(1+\sum_{i=1}^{m}\frac{c_i}{2^i}\right)=2^{m+2+e}\left(1+\sum_{i=1}^{m}\frac{c_i}{2^i}+\frac{1}{2^{m+2}}+\sum_{i=m+3}^{2m+2}\frac{b_i}{2^i}\right)$$

et le résultat est représenté par le nombre $2^{m+2+e}\left(1+\sum_{i=1}^{m}\frac{c_i}{2^i}\right)$ (même avec l'arrondi au plus proche), on a **perdu l'information sur le plus petit nombre**, c'est le phénomène d'absorption.

Différence de deux nombres proches : phénomène de cancellation Considérons par exemple les nombres $y = 1+\frac{1}{2^4}+\frac{1}{2^6}+\frac{1}{2^7}+\frac{1}{2^8}$, et $x = 1$. Voici leur représentation binaire exacte, ainsi que celle de $y - x$:

	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8
$y =$	1,	0	0	0	1	0	1	1	1
$x =$	1,	0	0	0	0	0	0	0	0
$y - x =$	0,	0	0	0	1	0	1	1	1

Dans une représentation des nombres réels avec une mantisse de 4 bits, on a $y_r = 2^0 \times 1,\underline{0001}$ et $x_r = 2^0 \times 1,\underline{0000}$ et donc $y_r - x_r = 2^{-4} \times 1,\underline{0000}$, alors que $y - x$ admet une représentation exacte $y - x = 2^{-4} \times 1,\underline{0111}$, on voit ainsi qu'il y a eu une **perte de chiffres significatifs**, c'est le phénomène de cancellation.

Test d'égalité à zéro De manière un peu équivalente, il est quasi impossible d'effectuer un test d'égalité à zéro pour une différence de deux réels, ou bien de tester l'égalité stricte entre deux réels, en raison des représentations approximatives de ces représentations. Par exemple le test $0.1+0.2 == 0.3$ (ou bien $0.3-0.2-0.1 == 0$) renvoie la valeur `False`. Pour comprendre cela, on peut raisonner avec une mantisse de 4 bits. On pose $x = 0.1$; $y = 0.2$ et $z = 0.3$. On a les représentations suivantes :

- pour x_r, E = −4, M = 1,1001 ;
- pour y_r, E = −3, M = 1,1001 ;
- pour z_r, E = −2, M = 1,0011.

Afin de faire la somme binaire, on écrit x_r et y_r avec le même exposant, soit par exemple $x_r = 2^{-4}(1+2^{-1}+2^{-4}) = 2^{-3}(2^{-1}+2^{-2}+2^{-5})$. La dernière écriture fait intervenir temporairement une mantisse avec $b_0 = 0$, et correspond à un décalage vers la droite des bits. On obtient pour la somme binaire des mantisses :

$$\begin{array}{rll} & 0\,1100 & x_r \text{décalé} \\ + & 1\,1001 & y_r \\ = & 2\,0101 & \end{array}$$

avec la première colonne qui correspond à b_0.

On obtient donc pour $x_r + y_r$, $2^{-3}(2+2^{-2}+2^{-4})$, soit en écriture normalisée $2^{-2}(1+2^{-3}+2^{-5})$, c'est à dire E = −2, M = 1,0010, le dernier bit n'étant pas représentable ici. On peut comparer avec l'écriture de z_r, E = −2, M = 1,0011. Les deux nombres ne correspondent pas.

On retrouve ce genre de comportement pour des mantisses stockées sur un nombre de bits m quelconque.

Conclusion On peut retenir des règles pratiques qui minimisent les erreurs lors des calculs avec les réels :

- Ne pas additionner des nombres de valeur absolue trop différentes.
- Ne pas soustraire des nombres trop proches.
- Ne pas utiliser le test d'égalité.

3. Exemples célèbres d'erreurs

3.1. Le crash d'Ariane 5

Le quatre juin 1996, c'est une erreur de programmation qui a causé le crash de la fusée européenne Ariane 5. Plus précisément, un réel codé sur 64 bits donnant la vitesse horizontale de la fusée était converti en un entier signé sur 16 bits. Or l'entier obtenu était plus grand que $32767 = 2^{15} - 1$, le plus grand entier représentable sur 16 bits. La conversion échouait donc. Tous les tests logiciels avaient pourtant réussi, mais ils avaient été effectués avec les données d'Ariane 4 pour laquelle la vitesse horizontale restait inférieure au maximum de 32767.

3.2. Missile Patriot

En février 1991, pendant la Guerre du Golfe, une batterie américaine de missiles Patriot, à Dharan (Arabie Saoudite), a échoué dans l'interception d'un missile Scud irakien. Le Scud a frappé un baraquement de l'armée américaine et a tué 28 soldats. La commission d'enquête a conclu à un calcul incorrect du temps de parcours, dû à un problème d'arrondi. Les nombres étaient représentés en virgule fixe sur 24 bits, donc 24 chiffres binaires. Le temps était compté par l'horloge interne du système en 1/10 de seconde. Malheureusement, 1/10 n'a pas d'écriture finie dans le système binaire : 1/10 = 0,1 (dans le système décimal) = 0,0001100110011001100110011... (dans le système binaire). L'ordinateur de bord arrondissait 1/10 à 24 chiffres, d'où une petite erreur dans le décompte du temps pour chaque 1/10 de seconde. Au moment de l'attaque, la batterie de missile Patriot était allumée depuis environ 100 heures, ce qui avait entraîné une accumulation des erreurs d'arrondi de 0,34 s. Pendant ce temps, un missile Scud parcourt environ 500 m, ce qui explique que le Patriot soit passé à côté de sa cible.

SOLUTIONS DES EXERCICES

Solution 1 Les entiers 13 et 11 ont pour représentations binaires respectives $(1101)_2$ et $(1011)_2$.

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

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

qui est bien l'écriture binaire de 143.

Solution 2

```
def dec2bin(n : int, N : int) -> list :
    assert 0 <= n < 2**N
    L = [0]*N
    i = N-1
    while n != 0 :
        L[i] = n%2
        n = n//2
        i -= 1
    return L
```

```
>>> dec2bin(22, 5)
[1, 0, 1, 1, 0]
```

Solution 3

1)

```
1 def bin2dec(L : list) -> int :
2     n = 0
3     N = len(L)
4     for k in range(N) :
5         n += L[N-k-1]*2**k
6     return n
```

```
>>> bin2dec([0, 1, 1, 0, 0, 1])
25
```

[Calcul de la complexité] On compte 3 opérations en lignes #2 et #3, et pour $i \in [\![1, N]\!]$ le i^e passage dans la boucle `for` (correspondant à $k_i = i-1$) donne $4+i$ opérations en ligne #5, donc :

$$C_n = 3 + \sum_{i=1}^{N} (4+i) = 3 + 4N + \frac{N(N+1)}{2} = \boxed{O(N^2)}.$$

2) 2.1) On a $n' = \sum_{k=0}^{N-2} c_{k+1} 2^k$, donc : $n = 2n' + c_0$.

2.2)

```
def bin2dec(L : list) -> int :
    if len(L) == 0:
        return 0
    else :
        return 2*bin2dec(L[:-1]) + L[-1]
```

```
>>> bin2dec([0, 1, 1, 0, 0, 1])
25
```

2.3) On a $C_0 = 1$ (pour le test) et si $N \geqslant 1$ on compte $C_N = 4 + C_{N-1}$ donc : $C_N = 3 + 4N = \boxed{O(N)}$.

2.4)

```
def bin2dec(L : list) -> int :
    n = 0
    for c in L :
        n = 2*n+c
    return n
```

```
>>> bin2dec([0, 1, 1, 0, 0, 1])
25
```

Solution 4 Les codages de 3 et -4 sur 8 bits sont respectivement 00000011 (on code directement 3) et 11111100 (on code $-4+256 = 252$).

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

qui représente bien -1 (on décode $n' = 255$ donc $n = n' - 256 = -1$).

$$\begin{array}{r} 1\,1\,1\,1\,1\,1\,0\,0 \\ \times\ 0\,0\,0\,0\,0\,0\,1\,1 \\ \hline \scriptstyle 1\ 1\ 1\ 1 \\ 1\,1\,1\,1\,1\,1\,0\,0 \\ +\ 1\,1\,1\,1\,1\,0\,0\,0 \\ \hline 1\,1\,1\,1\,0\,1\,0\,0 \end{array}$$

qui représente bien -12 (on décode $n' = 244$ donc $n = n' - 256 = -12$).

Solution 5

1.
```
def dec2binSigne(n : int, N : int) -> list :
    assert -2**(N-1) <= n < 2**N
    if n >= 0:
        return dec2bin(n, N)
    else :
        return dec2bin(n+2**N, N)
```
```
>>> dec2binSigne(-45, 8)
[1, 1, 0, 1, 0, 0, 1, 1]
```

2.
```
def bin2decSigne(L : list) -> int :
    if L[0] == 0:
        return bin2dec(L)
    else :
        return bin2dec(L)-2**len(L)
```
```
>>> bin2decSigne([1, 1, 0, 1, 0, 0, 1, 1])
-45
```

Solution 6

1) On a $x > 0$ et $1 \leqslant x < 2$ donc $s = 0$ et $\mathrm{E} = 0$. Pour déterminer M, on utilise l'algorithme décrit :

i	y_i	b_i
1	0.0625	0
2	0.125	0
3	0.250	0
4	0.50	1

On a donc $b_1 b_2 b_3 b_4 = 0001$, on a $x = (1,0001)_2$ en écriture binaire.

2) Pour $x = 0.1$, on a $x = 2^{-4} \times 1.6$, d'où $s = 0$, $\mathrm{E} = -4$, et pour déterminer M, on écrit :

i	y_i	b_i
1	0.6	1
2	0.2	0
3	0.4	0
4	0.8	1

d'où $b_1 b_2 b_3 b_4 = 1001$.

Enfin, pour $x = -0.3$, on a $s = -1$, $|x| = 0.3 = 2^{-2} \times 1.2$, d'où $\mathrm{E} = -2$ et on a :

i	y_i	b_i
1	0.2	0
2	0.4	0
3	0.8	1
4	0.6	1

d'où $b_1 b_2 b_3 b_4 = 0011$.

Solution 7

1) On a $\mathrm{E}_{\text{normalisé}} \in [\![-2^{10}+2; 2^{10}-1]\!] = [\![-1022; 1023]\!]$.

2) On a $x_{\max} = 2^{\mathrm{E}}_{\max} \times \mathrm{M}_{\max}$ avec $\mathrm{M}_{\max} \simeq 2$. On a donc $x_{\max} = 2^{p}_{\max}$ avec $p_{\max} \simeq 1024$. Numériquement cela doit dépasser la capacité de pas mal de calculatrices, on peut donc chercher le résultat sous la forme $2^{p}_{\max} = 10^{n+\varepsilon}$, d'où $n + \varepsilon = p_{\max} \dfrac{\ln(2)}{\ln(10)}$. On trouve $n = 308, \varepsilon = 0,25$, d'où $x_{\max} = 1,8 \times 10^{308}$.

3) On a de même $x_{\min} = 2^{\mathrm{E}}_{\min} \times \mathrm{M}_{\min}$, avec pour les nombres normalisés, $\mathrm{M}_{\min} = 1$, soit $x_{\min} = 2^{p}_{\min}$ avec $p_{\min} = -1022$. Numériquement, on obtient comme précédemment $x_{\min} = 2{,}2 \times 10^{-308}$.

Solution 8

1) On a $\delta\mathrm{M} = 2^{-m} = 2^{-52}$.

2) Avec $x = (-1)^{s} \times 2^{\mathrm{E}} \times \mathrm{M}$, $\delta x = 2^{\mathrm{E}-m}$.

Pour les plus petits nombres, $\mathrm{E} = -1022$, d'où $\delta x = 2^{-1074} \simeq 10^{-323} << 1$.

Pour les plus grands nombres, $\mathrm{E} = 1023$, d'où $\delta x = 2^{971} \simeq 10^{292} >> 1$.

Solution 9

1) On a directement $\varepsilon = \dfrac{\delta\mathrm{M}}{\mathrm{M}} = \dfrac{2^{-m}}{\mathrm{M}}$. Avec $1 \leqslant \mathrm{M} < 2$, on a donc $2^{-m-1} \leqslant \varepsilon < 2^{-m}$.

2) On a $2^{-52} \simeq 10^{-16}$. On a donc 16 chiffres significatifs correctement représentés, et ceci *dans tout l'intervalle des réels représentés.*