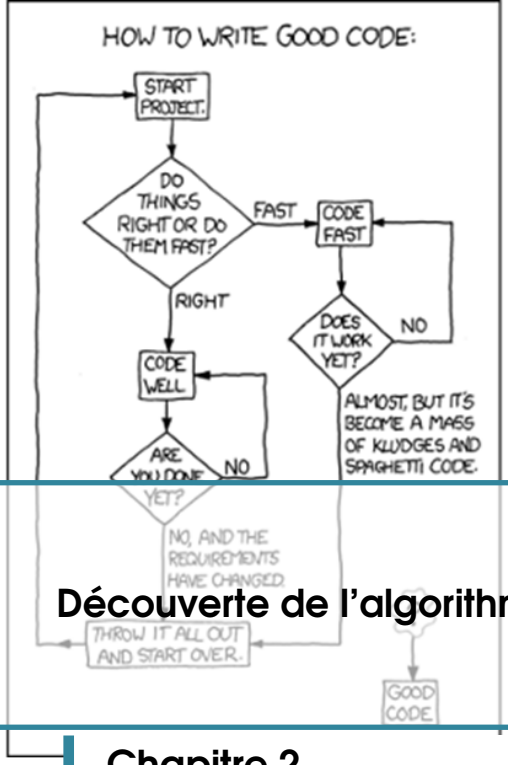

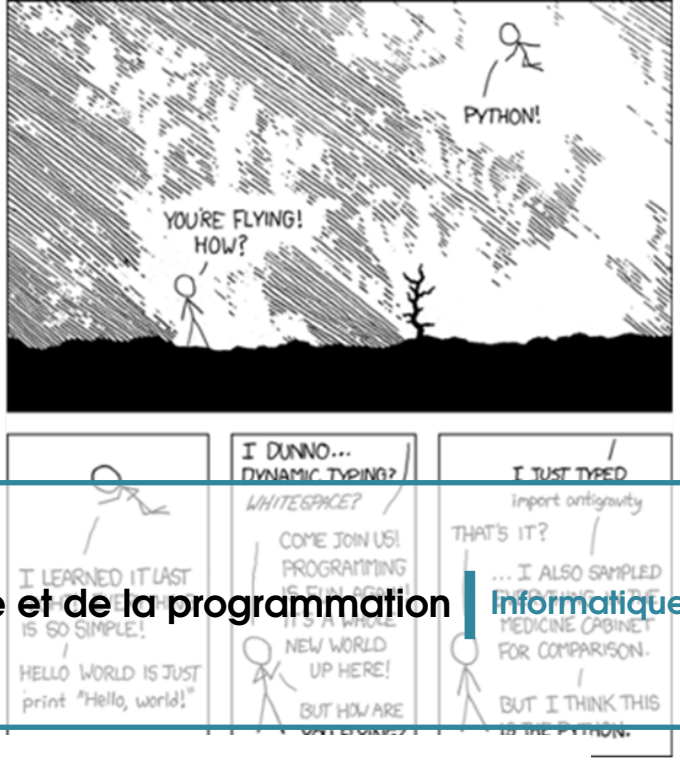

Chapitre 2

Représentation des nombres en mémoire

Cours

Savoirs et compétences :

- *Représentation des entiers positifs sur des mots de taille fixe.*
- *Représentation des entiers signés sur des mots de taille fixe.*
- *Entiers multi-précision de Python.*
- *Distinction entre nombres réels, décimaux et flottants.*
- *Représentation entre nombres réels, décimaux et flottants.*
- *Représentation des flottants sur des mots de taille fixe. Notion de mantisse, d'exposant.*
- *Précision des calculs en flottants.*

1 Bases de numération

Le système de numération courant est le système décimal (ou base 10). On décompose un entier en dizaines, centaines, milliers, etc. L'essentiel est alors qu'il y ait strictement moins de dix éléments dans chaque type de paquet. Ce nombre d'éléments peut être représenté par un chiffre. On écrit alors tous les chiffres à la suite. À gauche, on place les **chiffres de poids fort** (*most significant digit*). À droite, les **chiffres de poids faible** (*least significant digit*).

Ainsi 2735 représente deux milliers plus sept centaines plus trois dizaines plus cinq unités.

Définition Ecriture d'un nombre dans une base De manière générale :

$$\underline{a_n a_{n-1} \dots a_1 a_0}_{\,B} = \sum_{k=0}^{n} a_k B^k, \text{ et } \forall k \in [\![0;n]\!],\ a_k \in [\![0;B[\![.$$

On note B la base, a_k le chiffre de rang k.

▪ **Exemple** Décomposition de 247 en base 10 : $247_{(10)} = 2\cdot 10^2 + 4\cdot 10^1 + 7\cdot 10^0$.
Décomposition de 1001_2 en base 2 : $1001_2 = 1\cdot 2^{11_2} + 0\cdot 2^{10_2} + 0\cdot 2^{1_2} + 1\cdot 2^{0_2} = 1\cdot 2^{3_{10}} + 0\cdot 2^{2_{10}} + 0\cdot 2^{1_{10}} + 1\cdot 2^{0_{10}}$. ▪

1.1 Les systèmes de numération

1.1.1 Système décimal

Le système décimal est le système universellement utilisé. C'est la base de référence, ce qui signifie qu'un nombre est de manière implicite écrite en décimal, dès lors qu'il est écrit sans précision de base.

En python, les chiffres sont **affichés** par défaut dans le système décimal.

1.1.2 Système binaire

C'est la base de numération couramment utilisée en électronique ou informatique. C'est un système en base 2, l'écriture des nombres est donc composée des caractères de 0 et 1.

▪ **Exemple**

En binaire on compte de la manière suivante :

Base 10	Base 2
0	0000
1	0001
2	0010
3	0011
4	0100

Convertir $(11011001)_2$ *en base 10.*
Convertir $(42)_{10}$ *en base 2.*

▪

En Python, il est possible de manipuler des nombres en binaires en utilisant le préfixe `0b`. Ainsi, `0b111` est interprété comme l'entier 7. Il est aussi possible de convertir des entiers en binaires avec la fonction bin. Le résultat est une chaîne de caractère (`str`) de la forme `'0b111'`.

1.1.3 Système hexadécimal

Ce système à base 16 est le plus utilisé en électronique numérique car il permet une représentation compacte ce qui, dans les systèmes actuels à grande capacité mémoire, est un avantage non négligeable. Ainsi, les chiffres de ce système de numération sont 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

▪ **Exemple**

En hexadécimal on compte de la manière suivante :

Base 10	Base 2	Base 16	Base 2	Base 10	Base 16
0	0000	00	10	1010	0A
1	0001	01	11	1011	0B
2	0010	02	12	1100	0C
3	0011	03	13	1101	0D
4	0100	04	14	1110	0E
5	0101	05	15	1111	0F
6	0110	06	16	1 0000	10
7	0111	07	17	1 0001	11
8	1000	08	18	1 0010	12
9	1001	09	19	1 0011	13

Convertir $(BC5)_{16}$ en base 10.
Convertir 192 en base 16. ▪

En Python, il est possible de manipuler des nombres en hexadécimal en utilisant le préfixe `0x`. Ainsi, `0xA1` est interprété comme l'entier 162. Il est aussi possible de convertir des entiers en hexadécimal avec la fonction `hex`. Le résultat est une chaîne de caractère (`str`) de la forme `'0xa1'`.

1.2 Algorithme de décomposition d'un entier dans une base

Méthode La méthode générale pour décomposer un entier n en base k est la suivante :
- on effectue des divisions euclidiennes successives par k jusqu'à avoir un quotient nul ;
- on récupère les restes en remontant les calculs.

```
def decomposition(x:int, k:int)->str:
    """Donne la décomposition de l'entier x dans la base k.
    Entrées :
        x(int) : le nombre entier àécrire dans une nouvelle base
        k(int) la base de décomposition
    Sortie :
        le nombre dans la nouvelle base sous forme de chaine de caractères
    """
    n = x
    chaine = ''
    if n==0:
        chaine='0'
    while n > 0:
        r = n % k
        chaine = str(r) + chaine
        n = n // k
    return(chaine)
```

2 Représentation des entiers sur un ordinateur

2.1 Cadre

Définition Mot-machine En architecture informatique, un mot est une unité de base manipulée par un microprocesseur. On parle aussi de mot machine. La taille d'un mot s'exprime en bits. Elle est souvent utilisée pour classer les microprocesseurs (32 bits, 64 bits, etc.).

Sur un ordinateur on travaille sur des mots-machine de 64 bits (8 octets).

Les opérations d'addition et de multiplication d'entiers internes au processeur se font sur 64 bits. De manière générale, on s'intéressera au fonctionnement sur des ordinateurs travaillant sur des mots de n bits ($n \geq 2$).

Par ailleurs, sur les disques durs, les donées sont stockées sous forme binaire. Ainsi, un octet = 8 bits, un kilo octet = 1000 octet *etc.*

2.2 Représentation des entiers

Les entiers sont directement codés en binaire. Lorsque les mots-machines sont codés sur n bits, il est alors possible de coder des nombres compris entre 0 et $2^n - 1$.

R

- On peut alors observer des dépassements de capacité : si les mots sont codés sur 4 bits, $(1100)_2 + (1000)_2 = (0100)_2$ (et non $(10100)_2$). Dans ce cas, la retenue est perdue.

- En Pyton, les entiers ne sont pas codés sur un nombre de bits défini. On est seulement limité par la mémoire de la machine.

2.3 Représentation des entiers relatifs – Complément à 2

Lorsque les mots sont codés sur n bits et qu'on souhaite coder des entiers positifs et négatifs, on pourra coder des nombres compris dans l'intervalle $[\![-2^{n-1}, 2^{n-1}-1]\!]$. Les nombres positifs seront codés en binaire comme vu précédemment.

Les nombres négatifs seront codés selon la méthode du complément à 2.

Méthode Si les mots sont codés sur n bits, si on souhaite coder $a<0$, a sera codé comme l'entier naturel $a+2^n$.

▪ **Exemple** Soit un système où les entiers relatifs sont codés sur 3 bits. On peut donc coder des nombres compris dans $[\![-4,3]\!]$.

- 2 est codé par $2=(010)_2$;
- -2 est codé par $-2+2^3=6=(110)_2$.

Remarques :

- les nombres négatifs commencent donc par 1 ;
- pour convertir $a=(110)_2$ en base 10 lorsque le codage est sur 3 bits, on a : $a=(110)_2-2^3=6-8=-2$.

▪

3 Représentation des nombres réels

3.1 Conversion de la partie fractionnaire des nombres réels

En notation décimale, les chiffres à gauche de la virgule représentent des entiers, des dizaines, des centaines, etc. et ceux à droite de la virgule, des dizièmes, des centièmes, des millièmes, etc.

▪ **Exemple**

$$3,3125_{(10)}=3\cdot 10^0+3\cdot 10^{-1}+1\cdot 10^{-2}+2\cdot 10^{-3}+5\cdot 10^{-4}$$

▪

Par analogie, pour écrire un nombre binaire à virgule, on utilise les puissances négatives de 2.

▪ **Exemple**

$$\begin{aligned} 11,0101_{(2)} &= 1\cdot 2^1+1\cdot 2^0+0\cdot 2^{-1}+1\cdot 2^{-2}+0\cdot 2^{-3}+1\cdot 2^{-4}\\ &= 2+1+0+0,25+0+0,0625\\ &= 3,3125_{(10)} \end{aligned}$$

Le codage de la partie entière (3 dans l'exemple précédent) ne pose pas de problèmes particuliers. Pour la partie fractionnaire (0,3125), il est nécessaire d'adapter la procédure. ▪

Méthode **Conversion d'une partie fractionnaire en binaire**

1. On multiplie la partie fractionnaire par 2.
2. La partie entière obtenue représente le poids binaire (limité aux seules valeurs 0 ou 1).
3. La partie fractionnaire restante est à nouveau multipliée par 2.
4. On procède ainsi de suite jusqu'à ce qu'il n'y ait plus de partie fractionnaire ou que le nombre de bits obtenus correspond à la taille du mot mémoire dans lequel on stocke cette partie.

▪ **Exemple** *Conversion de la partie fractionnaire 0,3125*

0,3125	x	2	=	0,625	=	0	+	0,625
0,6250	x	2	=	1,250	=	1	+	0,250
0,2500	x	2	=	0,500	=	0	+	0,500
0,5000	x	2	=	1,000	=	1	+	0,000

On considère les parties entières de haut en bas : $0,3125_{(10)} = 0,0101_{(2)}$. ▪

▪ **Exemple** *Convertir la partie fractionnaire 0,1* ▪

3.2 Représentation d'un nombre réel en flottant en utilisant la norme IEEE-754

Pour représenter des réels, nombres pouvant être positifs, nuls, négatifs et non entiers, on utilise la représentation en virgule flottante (*float* en anglais) qui fait correspondre au nombre 3 informations :

$$-243,25_{(10)} = \underbrace{-}_{1} \; \underbrace{0,24325}_{2} \cdot 10^{\overbrace{3}^{3}}$$

On appelle alors :
1. le signe (positif ou négatif) ;
2. la mantisse (nombre de chiffres significatifs) ;
3. l'exposant : puissance à laquelle la base est élevée.

Sous cette forme normalisée, il suffit de mémoriser le signe, l'exposant et la mantisse pour avoir une représentation du nombre en base 10. Il n'est pas utile de mémoriser le 0 avant la virgule puisque tous les nombres vont commencer par 0. En faisant varier l'exposant, on fait « flotter » la virgule décimale.

Pour représenter un nombre réel dans la machine, il faut préalablement écrire les nombres sous la forme (norme IEEE 754 – Institute of Electrical and Electronics Engineers) :

$$\text{signe 1, mantisse x } 2^{\text{exposant}}.$$

▪ **Exemple** En binaire, on a $-100110,10101 = -1,0011010101 \times 2^5$ avec $-$ le signe, 0011010101 la mantisse et 5 l'exposant. ▪

Le tableau décrit la répartition des bits selon le type de précision : la taille de la mantisse (m bits) donne la précision mais suivant la valeur de l'exposant, la précision sera totalement différente.

	Signe	Exposant	Mantisse
Simple précision – 32 bits	1	8	23
Double précision – 64 bits	1	11	52
Précision étendue – 80 bits	1	15	64

Méthode
1. Convertir en binaire les partie entière et fractionnaire du nombre sans tenir compte du signe.
2. Décaler la virgule vers la gauche pour le mettre sous la forme normalisée (IEEE 754).
3. Codage du nombre réel avec les conventions suivantes :
   - signe = 1 : Nombre négatif (Signe = 0 : Nombre positif) ;
   - le chiffre 1 avant la virgule étant invariant pour la forme normalisée, il n'est pas codé ;
   - on utilise un exposant décalé au lieu de l'exposant simple (complément sur octet). Ainsi, on ajoute à l'exposant simple la valeur 127 en simple précision et 1023 en double précision (c'est à dire $2^{n-1}-1$ où n est le nombre de bits de l'exposant) ;
   - la mantisse est complétée à droite avec des zéros.

▪ **Exemple** On désire représenter le nombre - 243,25 en virgule flottante au format simple précision.
1. $243,25_{(10)} = 11110011,01_{(2)}$
2. $243,25_{(10)} = 1,111001101_{(2)} \times 2^7$: décalage de 7 bits vers la gauche
3. Exposant décalé : $7 + 127 = 134_{(10)} = 1000\,0110_{(2)}$ sur $n = 8$ bits
4. 111001101 00000000000000

▪

S	Exposant								Exposant																							
1	1	0	0	0	0	1	1	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C				3				7				3				4				0				0				0				

3.3 Compléments sur la norme IEEE 754

On interprète donc la suite de bits $s e_{10} \dots e_0 m_1 \dots m_{52}$ comme le nombre x défini comme suit. Notons $e = \underline{e_{10} \dots e_0} = \sum_{k=0}^{10} e_k 2^k$.

- Si $e \in [\![1, 2047[\![$, x est le nombre *normalisé* : $x = s \times \underline{1, m_1 \dots m_{52}} \times 2^{\left(-1023+\underline{e_{10}\dots e_0}\right)} = s \times \left(1 + \sum_{k=1}^{52} \frac{m_k}{2^k}\right) \times 2^{\left(-1023+\sum_{k=0}^{10} e_k 2^k\right)}$.
- Si $e = 0$ et $m_1 = \dots = m_{52} = 0$: $x = 0$ (deux versions : $+0$ et -0).
- Si $e = 0$ et $m_1, \dots, m_{52}$ non tous nuls, x est le nombre *dénormalisé* : $x = s \times \underline{0, m_1 \dots m_{52}} \times 2^{-1022} = s \times \left(\sum_{k=1}^{52} \frac{m_k}{2^k}\right) \times 2^{-1022}$.
- Si $e = 2047$ et $m_1 = \dots = m_{52} = 0$: $x = s\infty$ ($+\infty$ ou $-\infty$).
- Si $e = 2047$ et $m_1, \dots, m_{52}$ non tous nuls : $x =$ NaN.

R On ne rentrera pas dans le détail des signification de $+\infty$, $-\infty$ et de NaN.

Les nombres normalisés permettent de représenter de façon précise les réels de $[-M, -m] \cup [m, M]$ avec $m \approx 2^{-1022} \approx 2 \times 10^{-308}$ et $M \approx 2^{1024} \approx 1,8 \times 10^{308}$.

Les nombres dénormalisés ne respectent pas la convention de la notation scientifique standard, mais permettent de représenter des nombres plus petits que les nombres normalisés ne peuvent.

3.4 En Python

Avec Python, on peut accéder à la représentation d'un nombre flottant par la méthode `.hex()`. Attention, le nombre est écrit en hexadécimal.

▪ **Exemple** Avec `5.5`.

```
>>>5.5.hex()
'0x1.6000000000000p+2'
```

▪

3.5 Problèmes de précision

On rencontre différents types de problèmes de précision.

1. Les problèmes liés aux arrondis des calculs.
2. Les problèmes liés au passage à la représentation binaire.

3.5.1 Problèmes liés aux arrondis

Supposons que l'on veuille effectuer des calculs avec des chiffres décimaux n'ayant que deux chiffres après la virgule.

▪ **Exemple** Pour la multiplication : $1,23 \times 1,56 = 1,9188$ ▪

▪ **Exemple** Pour l'addition : $1,23 \times 10^3 + 4,56 \times 10^0 = 1,23456 \times 10^3$ ▪

Pour garder deux chiffres après la virgule, on arrondit le résultat et l'on introduit donc une erreur d'approximation.

Ce problème se pose en décimal, comme en binaire !

3.5.2 Problèmes liés au passage à la représentation binaire

Attention : Les représentations binaires et décimales partagent les *mêmes* problèmes d'arrondis. Cependant, on crée des erreurs d'arrondis lors du *passage* d'une représentation à l'autre.

▪ **Exemple** En Python, on rentrera dans la console des nombres en écriture décimale mais le calcul interne se fera en binaire. Cela donne la chose suivante.

```
>>> 0.1+0.2 == 0.3
       False
>>> 0.1+0.2
       0.30000000000000004
>>> 0.1+0.2-0.3
       5.551115123125783e-17
```

▪

En conséquence, on ne teste jamais l'égalité entre deux flottants.

Application 01

Applications – Bases

Savoirs et compétences :
❒ .

Exercice 1 –

Question 1 *Réalisez la conversion des nombres suivants dans les autres systèmes de numération :*

1. $(10050)_{(10)}$
2. $(10010001)_{(2)}$
3. $(A3F)_{16}$

Exercice 2 – On désire utiliser 12 bits pour comptabiliser des objets.

Question 1 *Quel est le nombre maximum d'objets qu'il est possible de compter ?*

Question 2 *Indiquer le numéro du premier et du dernier (dans les systèmes de numération décimale, binaire et hexadécimale).*

Exercice 3 – On désire compter 65000 objets.

Question 1 *Sur combien de bit peut-on réaliser cette opération ?*

Question 2 *Quel est le premier et le dernier nombre (dans les systèmes de numération binaire et hexadécimale) ?*

Exercice 4 – Soit une machine où les nombres entiers sont codés sur 8 bits.

Question 1 *Donner le plus grand et le plus petit nombre représentable selon que le codage utilisé est non signé ou signé.*

Question 2 *Écrire dans les différents formats signés les nombres décimaux 1, −1, 111 et 55.*

Question 3 *Quelles sont inversement les valeurs décimales codées par $4C$ et $B4$ suivant les différents codages signés et celui non signé.*

Exercice 5 –

Question 1 *Effectuez les opérations arithmétiques suivantes dans les systèmes de numération binaire (codé sur 8 bits) :*

1. $71+35=$
2. $15-25=$
3. $121-75=$
4. $-51-77=$

Exercice 6 –

Question 1 *Écrire dans le format flottant simple précision (IEEE 754) les nombres 1,0 ; −1,0 ; 15,25 et −3,26. Les résultats seront donnés en hexadécimal.*