L. n°3

CODAGE DE L'INFORMATION
Représentation des nombres

Un peu d'histoire
Jusqu'à la fin du moyen-age, on utilisait en Europe un système visécimal (ou vigésimal), utilisant la base 20. A cette époque, les nombres n'utilisaient pas encore les chiffres arabes.
On retrouve des traces de ce système dans la langue française (quatre-vingt). L'hôpital des quinze-vingts, fondé en 1260 à Paris pouvait accueillir 300 patients.

Dans ce chapitre, nous allons nous intéresser à la façon dont un nombre (entier ou réel) peut être représenté à l'intérieur d'un ordinateur. En effet, la mémoire des ordinateurs est découpée en blocs de 8 bits (un octet). Par exemple, un processeur de 64 bits disposera de 64 bits pour représenter un nombre.

I. Représentation des entiers naturels

1. Cas général : base *b*

Depuis la fin du Moyen Âge, nous comptons en **base** 10. Cela signifie que nous utilisons 10 symboles différents, appelés chiffres : 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9. On fait des "paquets de 10" pour passer d'une unité à la suivante. On utilise alors le tableau de décomposition suivant :

Puissance	Dix-mille	Milliers	Centaine	Dizaine	Unité
de					
12 563					

Par exemple, le nombre 12 563 se décompose alors sous la forme :

12 563 = ...

12 563 = ...

12 563 = ...

On note alors : 12 563 = ... = ...

De manière plus générale, un nombre peut s'écrire dans une base ***b*** quelconque.

Définition : Une **base** d'un système de numération positionnel est . . . . . supérieur ou égal à 2. Un nombre en base ***b*** s'écrit alors sous la forme $(c_n\ c_{n-1}\ \ldots\ c_2\ c_1\ c_0)_b$ où les chiffres c_i sont tous . . . . . .

En informatique, les bases les plus utilisées sont la base 2, ou **binaire** (les bits), la base 16, ou **hexadécimale**, (adresses mémoire, couleurs HTML) et parfois aussi la base 8, ou **octale** (droits des fichiers UNIX).

2. Système binaire

a. Définition

Ce système dit de base 2 comprend . . . . . .

Chacun d'eux est aussi appelé . . . . . .

Par analogie, on utilise alors le tableau de décomposition suivant :

. . . . .	. . . . .	. . . . .	. . . . .	. . . . .	. . . . .

Exemple : N = $(10110)_2$. Ce nombre N peut être écrit sous la forme suivante :
N = $(10110)_2$ = ...

On note alors : $(10110)_2$ = ... = ...

b. Opérations

- **Addition en base 2**

On veut additionner les deux nombres $(101101)_2$ et $(1001)_2$.
Il y aura des retenues lorsque la somme de deux bits sera égale 2.

		1			1	
	1	0	1	1	0	1
+			1	0	0	1
	1	1	0	1	1	0

Ainsi on a : $(101101)_2 + (1001)_2 = (110110)_2$

- **Multiplication en base 2**

On veut multiplier les deux nombres $(1101)_2$ et $(101)_2$.
Il y aura des retenues lorsque la somme de deux bits sera égale 2.

		1			1	
			1	1	0	1
				1	0	1
1	1	1	1			
			1	1	0	1
+		0	0	0	0	
+	1	1	0	1		
1	0	0	0	0	0	1

Ainsi on a : $(1101)_2 + (101)_2 = (1000001)_2$

3. Système hexadécimal

Ce système dit de base 16 comprend . . . . . .

On utilise les symboles suivants : . . . . . .

Exemple : On considère le nombre N écrit en base 16 tel que : N= $(AC53)_{16}$.
Ce nombre N peut se décomposer sous la forme suivante :

N = $(AC53)_{16}$ = ...

On note alors : $(AC53)_{16}$ = ... = ...

4. Correspondance entre nombres de différentes bases

Décimal	Binaire	Hexadécimal	Décimal	Binaire	Hexadécimal
0	0	0	8	1000	8
1	1	1	9	1001	9
2	10	2	10	1010	A
3	11	3	11	1011	B
4	100	4	12	1100	C
5	101	5	13	1101	D
6	110	6	14	1110	E
7	111	7	15	1111	F

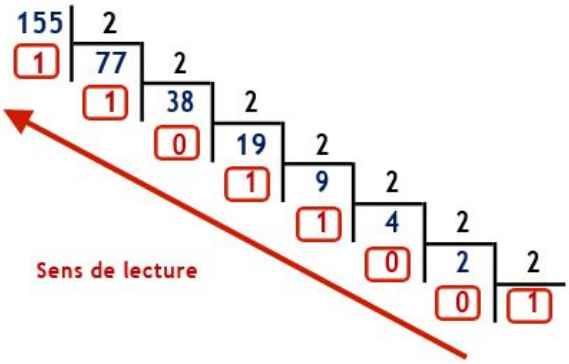
II. Changements de base

1. Conversion d'un nombre décimal en un nombre d'une autre base

Méthode : diviser le nombre décimal à convertir par la base *b* et conserver le reste de la division.

Le quotient obtenu est divisé par *b* et conserver le reste. Il faut répéter l'opération sur chaque quotient obtenu jusqu'à ce que le dernier quotient soit nul.
Les restes successifs sont écrits, en commençant par le dernier, **de la gauche vers la droite** pour former l'expression de $(N)_{10}$ dans le système de base *b*. Cette méthode est dite « ***Méthode des divisions successives*** ».

Exemple 1: conversion d'un nombre décimal en base 2 : On veut convertir $(155)_{10}$ en base 2.
Méthode : on divise le nombre décimal 155 par 2.
On obtient la suite successive de divisions suivante :



$155 = (10011011)_2$

Exemple 2: conversion d'un nombre décimal en base 16 : On veut convertir $(175)_{10}$ en base 16.
Méthode : on divise le nombre décimal 175 par . . . . . .
On obtient la suite successive de divisions suivante :

Réponse : $(175)_{10} = (. . . . . .)_{16}$

2. Conversion d'un nombre hexadécimal en binaire

Chaque symbole du nombre écrit dans le système hexadécimal est remplacé par son équivalent écrit dans le système binaire.
Exemple : Convertir $N = (ECA)_{16} = (1110\ 1100\ 1010)_2$

1110	1100	1010
E	C	A

3. Conversion d'un nombre binaire en hexadécimal

C'est l'inverse de la précédente. Il faut donc regrouper les 1 et les 0 du nombre par 4 en commençant par la droite, puis chaque groupe est remplacé par le symbole hexadécimal correspondant.
Exemple : Convertir $N = (1\ 1000\ 0110\ 1111)_2 = (\ 1\ 8\ 6\ F\ )_{16}$

1	8	6	F
0001	1000	0110	1111

4. Quelques définitions

- **Bit :**
  Le bit est une unité élémentaire d'information ne pouvant prendre que deux valeurs distinctes *(notées 0 ou 1)*.

- **Octet (ou byte en anglais)**

Un octet est composé de 8 bits :

On distingue :

1	1	0	1	0	1	0	1
b7	b6	b5	b4	b3	b2	b1	b0

- Le bit de poids fort b7 (*MSB : Most Significant Bit*)
- Le bit de poids faible b0 (*LSB : Least Significant Bit*)

- **Kilo-octet (Koctet)**

Un Kilo-octet est composé de 1024 octets ($2^{10} = 1024$).

- **Quelques valeurs à retenir :**

• Un kilooctet (ko) = 10^3 octets
• Un mégaoctet (Mo) = 10^6 octets
• Un gigaoctet (Go) = 10^9 octets
• Un téraoctet (To) = 10^{12} octets
• Un pétaoctet (Po) = 10^{15} octets

III. Représentation des nombres

1. Codage en binaire - nombre de combinaisons

En informatique, dans une machine, toutes les informations sont codées sous forme d'une suite de "0" et de "1". Tous les caractères utilisables par l'ordinateur sont codés en binaire sur un octet.

Avec 1 bit, on peut écrire . . . . combinaisons : . . . . . ou . . . . . .

Avec 2 bits, on peut écrire . . . . combinaisons : . . . . . .

Avec 3 bits, on peut écrire . . . . combinaisons : . . . . . .

Avec 4 bits, on peut écrire . . . . combinaisons : . . . . . .

Avec *k* bits, on peut écrire . . . . combinaisons : . . . . . .

Ainsi, avec un octet (soit . . . . bits), on peut écrire : . . . . . .

2. Les entiers naturels

Le codage d'un entier par un ordinateur se fait en général sur 32 bits ou 64 bits.

Un entier naturel est un nombre entier positif ou nul.
Pour coder des nombres entiers naturels compris entre 0 et 255, il nous suffira de 8 bits (un octet) car $2^8 = 256$.
D'une manière générale un codage sur n bits pourra permettre de représenter des nombres entiers naturels compris entre 0 et $2^n - 1$.

3. Les entiers relatifs

Un entier relatif est un entier naturel muni d'un signe positif ou négatif.

Pour coder un entier relatif en binaire, on utilise une représentation dans laquelle le bit de poids le plus fort sert à représenter son signe. Ainsi le bit de poids fort vaut :

- 0 pour un entier positif
- 1 pour un entier négatif.

Les autres bits représentent la **valeur absolue** du nombre en binaire.
On doit donc imposer le format (par exemple 8 bits ou 16 bits) afin de connaître le bit de signe.
On parle parfois de ***nombres signés***.

Dans le cas d'un nombre négatif, on utilise la **méthode du complément à deux** en suivant les étapes ci-dessous :

1. Représenter en binaire la valeur absolue de ce nombre ;
2. Faire le ***complément à 1*** de ce nombre (on remplace les 0 par des 1 et les 1 par des 0) ;
3. Ajouter 1 au nombre en binaire obtenu à l'étape 2.

Exemple : On veut représenter -3 sur 8 bits

1. Coder 3 en binaire	
2. Complément à 1	
3. Ajouter 1	
Codage obtenu :	

On obtient donc : $(-3)_{10} = (\ldots\ldots\ldots\ldots\ldots)_2$

Remarques :

Sur 8 bits, on pourra donc représenter les nombres "signés" de : . . . . . .

Soit les nombres entre . . . . . .

Sur 16 bits, on pourra donc représenter les nombres "signés" de : . . . . . .

Soit les nombres entre . . . . . .

4. Les nombres à virgule

a. Méthode

Un nombre à virgule (soit décimal, soit réel) est composé d'une partie entière et d'une partie fractionnaire après la virgule.

On considère un nombre écrit en base ***b*** de la façon suivante : $c_n\, c_{n-1} \ldots c_1\, c_0\, ,\, d_1\, d_2 \ldots d_k$

On obtient alors l'écriture décimale de ce nombre en effectuant l'opération suivante :

$$c_n \times b^n + c_{n-1} \times b^{n-1} + \ldots + c_1 \times b^1 + c_0 \times b^0 + d_1 \times b^{-1} + d_2 \times b^{-2} + \ldots + d_k \times b^{-k}$$

Exemples :

- $1011{,}1011 = 1\times2^3 + 1\times2^1 + 1\times2^0 + 1\times2^{-1} + 1\times2^{-3} + 1\times2^{-4} = 8 + 2 + 1 + 0{,}5 + 0{,}125 + 0{,}0625$
  Soit la correspondance suivante : $(1011{,}1011)_2 = (11{,}6875)_{10}$

- $(B9{,}2C)_{16} = B\times16^1 + 9\times16^0 + 2\times16^{-1} + C\times16^{-2} = 11\times16 + 9 + 2\times16^{-1} + 11\times16^{-2}$
  Soit la correspondance suivante : $(B9{,}2C)_{16} = (185{,}16796875)_{10}$
  On remarque dans cet exemple que la partie décimale est une valeur approchée seulement.

Pour convertir un nombre écrit en base 10 dans la base 2, on utilise la méthode suivante :

- On convertit en binaire la partie entière ;
- On effectue une suite de multiplication par 2 de la partie fractionnaire uniquement.

Exemple : convertir $(9{,}14)_{10}$ en binaire :

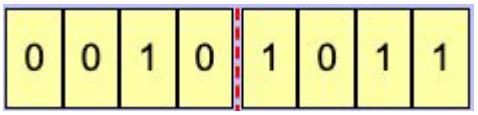
- Partie entière : $(9)_{10} = 1\times2^3 + 1\times2^0 = (1001)_2$
- Partie fractionnaire :
  $0{,}14 \times 2 = 0{,}28 = \mathbf{0} + 0{,}28$
  $0{,}28 \times 2 = 0{,}56 = \mathbf{0} + 0{,}56$
  $0{,}56 \times 2 = 1{,}12 = \mathbf{1} + 0{,}12$
  $0{,}12 \times 2 = 0{,}24 = \mathbf{0} + 0{,}24$ ….

Ainsi, on obtient $(1001,\mathbf{0010})_2$.

b. Virgule fixe

On parle de représentation à virgule fixe lorsqu'on décide par avance de la position de la virgule dans le codage. Par exemple sur un octet, on choisit le format "4,4", cela signifie que les 4 bits les plus à gauche correspondent à la partie entière et les 4 bits le plus à droite à la partie fractionnaire.

Par exemple :



= 2,6875

Remarque : cette représentation à virgule fixe nécessite de connaître à l'avance (lors de l'écriture du programme) l'ordre de grandeur des nombres à manipuler afin de positionner au mieux la virgule, ce qui n'est pas toujours possible. Pour pallier à ce manque de flexibilité, le concept de virgule flottante a été introduit.

c. Virgule flottante

Pour éviter de perdre de l'espace dans le format par la virgule fixe, on utilise la méthode d'**écriture à virgule flottante.** Dans les machines, les nombres sont alors appelés des flottants (floats en anglais).

La norme IEEE 754 est la norme la plus employée pour la représentation des nombres à virgule flottante dans le domaine informatique.

En général**,** les données sont représentées en binaire sur 16 ou 32 bits en simple précision, et 64 bits en double précision.

Nous allons observer le format dit "**simple précision**" seulement cette année.
Ce format "simple précision" utilise 32 bits de la façon suivante :

- **1 bit de signe** (1 si le nombre est négatif et 0 si le nombre est positif)
- **8 bits pour l'exposant** : les valeurs peuvent alors varier de . . . . . .

  Cependant, les valeurs −127 et 128 seront réservées pour des cas spéciaux.

  Pour éviter la méthode du complément à 2 pour coder l'exposant, on fait le choix de ***biaiser*** cet exposant en lui ajoutant $2^7 - 1 = 127$, revenant donc ainsi à coder un entier positif.

- **23 bits pour la mantisse**.

Par exemple, on a le codage suivant :

Signe *1 bit*	**Exposant** *8 bits*	**Partie fractionnaire de la mantisse** *23 bits*
0	1000 0001	110 0000 0000 0000 0000 0000

Exemple : Codons le nombre décimal −123,375 :

- Ce nombre est négatif, le bit de signe sera : 1
- On code en binaire la valeur absolue : $(123{,}375)_{10}$ = ...
  Ensuite, nous décalons la virgule vers la gauche, de façon à ne laisser qu'un 1 sur sa gauche, on obtient la représentation suivante : . . . . . × 2 ····
  Cette notation est une notation de nombre flottant ***normalisé***.
  La mantisse est la partie à droite de la virgule, remplie de 0 vers la droite pour obtenir 23 bits.

  On obtient : *m* = ...

- Le décalage est égal à . . . . . , et nous devons le ***biaiser*** puis le convertir en binaire :

  . . . . . + 127 = ... codé par . . . . . .

  On a donc −118,625 qui est codé par . . . . . .

Les codages spéciaux :

• **Cas du 0 :** 0 ne peut pas avoir une mantisse commençant par un 1, il n'est donc pas possible de le représenter. Par convention, il a été décidé qu'un nombre vaut 0 si, et seulement si, tous les chiffres de son exposant et sa mantisse valent 0 (il y a donc +0 et −0) ;
• Les **NaN** (*not a number*) sont là pour signaler une erreur de calcul, comme une division par 0 ou la racine carrée d'un nombre négatif : ils sont représentés avec tous les chiffres de l'exposant égaux à 1 et une mantisse non nulle.

5. Les limites du codage des nombres

Nombre représentable	Signe	Exposant	Mantisse	Valeur approchée
Le plus grand	0	1111 1110	111 1111 1111 1111 1111 1111	$3{,}40282346 \times 10^{38}$
Le positif le plus proche de 0	0	0000 0001	000 0000 0000 0000 0000 0000	$1{,}17549435 \times 10^{-38}$
Le négatif le plus proche de 0	1	0000 0001	000 0000 0000 0000 0000 0000	$-1{,}17549435 \times 10^{-38}$
Le plus petit	1	1111 1110	111 1111 1111 1111 1111 1111	$-3{,}40282346 \times 10^{38}$

Exemple :
Si on essaie de comparer la somme des nombres 0,1 et 0,2 au résultat attendu 0,3, on constate que :

```
>>> 0.1+0.2 == 0.3
False
>>> 0.1+0.2
0.30000000000000004
>>>
```