Représentation des données

[Mise à jour le 5/10/2022]

Sources

- Numérique et sciences informatiques 1er - ellipses

1. Introduction

Les données et les programmes stockés dans la mémoire des machines numériques sont représentés à l'aide de deux chiffres : **0** et **1**.

> **0** et **1** sont appelés chiffres binaires (*Binary Digits*) ou **bits**.

Ces chiffres binaires sont regroupés en paquets de **8 bits** (***octets*** ou ***bytes***) puis couramment organisés en **mots** de 2, 4 ou 8 octets. Une machine dite 32 bits manipule des mots de 4 octets lorsqu'elle effectue des opérations. Ce regroupement de bits est à la base de la représentation des entiers, des réels et des caractères.

> Il est nécessaire d'utiliser des codes (**encoder**) pour représenter des **entiers**, des **décimaux** et des **caractères**.

2. Encodage des entiers naturels

2.1 Écriture en base 2

Une séquence de chiffres binaires peut s'interpréter comme un nombre écrit en base 2. Dans cette base, les chiffres **0** et **1** d'une séquence sont associés à un **poids** $\mathbf{2^i}$ qui dépend de la position i des chiffres dans la séquence.

> Une séquence de **n bits** b_i, $b_{n-1},b_{n-2},...,b_1,b_0$ correspond au nombre décimal N tel que :
> $$N_{10} = b_{n-1}2^{n-1}+b_{n-2}2^{n-2}+ ... +b_12^1+b_02^0 = \displaystyle\sum_{i=0}^{i=n-1}b_i2^i$$

ATTENTION

Cet encodage permet de représenter des entiers dans l'intervalle [**0**,**2^n - 1**]

Exemples
- Un octet (n = 8) permet de coder tous les nombres entre 0 et 255
- $10011011_2 = 1*2^7 + 0*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 1*2^0 = 128 + 16 + 8 + 2 + 1 = 155_{10}$

2.2 Écriture en base 16

La base 16 ou hexadécimale est souvent utilisée pour simplifier l'écriture des nombres binaires.

On passe d'un nombre en base 16 à un nombre en base 2 en regroupant les chiffres binaires par 4.

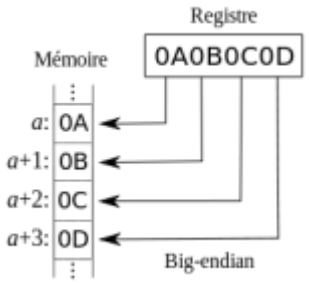
Exemple $1001\ 1110\ 0111\ 0101_2 = 9E75_{16}$

2.3 Boutisme

Sources

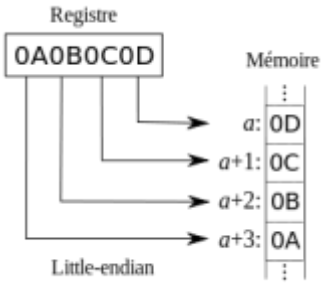
- Boutisme sur Wikipédia

La représentation en machine des entiers naturels sur des mots de 2, 4 ou 8 octets se heurte au problème de l'ordre dans lequel ces octets sont organisés en mémoire. Ce problème est appelé le **boutisme** (ou endianness).



- **Big endian**

Le gros boutisme consiste à placer l'**octet de poids fort** en premier, c'est-à-dire à l'adresse mémoire la plus petite.



- **Little endian**

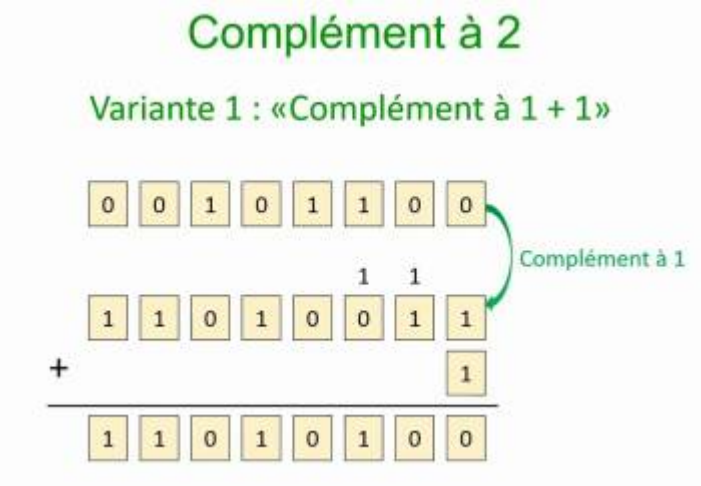
> Le petit boutisme consiste à placer l'**octet de poids faible** en premier. C'est-à-dire à l'adresse mémoire la plus petite.

> Bien que le boutisme soit géré au niveau du système d'exploitation, il peut être nécessaire d'en tenir compte lorsqu'on accède aux **octets en mémoire** ou lors d'**échanges sur un réseau**.

2.4 Bases 2,10 et 16 en Python

Voir Variables, types numériques et entrées / sorties dans la console

3. Encodage des entiers relatifs (complément à 2)



Ressource

- La notation en complément à 2 sur Wikipédia et SlidePlayer

*Le code complément à 2 permet d'effectuer simplement des **opérations arithmétiques** sur les entiers relatifs codés en binaire. Il opère toujours sur des nombres binaires ayant le **même nombre de bits** (par commodité, celui-ci est généralement un multiple de 4).* Wikipédia

> **ATTENTION**
>
> Le code complément à 2 permet de représenter des entiers dans l'intervalle [$\mathbf{-2^{n-1}}$,$\mathbf{2^{n-1} - 1}$]. Le bit de **poids fort** (bit le plus à gauche) ou **MSB** (Most Significant Bit) donne le **signe** du nombre représenté (**0 ⇒ positif** ou **1 ⇒ négatif**).

Exemple
- Un octet (n = 8) permet de coder en complément à 2 tous les nombres entre -128 et 127.
- Le nombre 01110001_2 est positif et égal à 89_{10}
- Le nombre 11111111_2 est négatif et égal à -1_{10}

Obtention d'un nombre binaire en complément à 2

> Avant de convertir un nombre décimal en binaire complément à 2, il faut s'assurer de représenter ce nombre avec suffisamment de bits. $\mathbf{-2^{n-1} \leq -N}$ avec N>0.

Exemple : nombre de bits nécessaires pour coder $\mathbf{-125000_{10}}$
$-2^{n-1} \leq -125000$
$\Rightarrow 2^{n-1} > 125000$
$\Rightarrow \log_{10}(2^{n-1}) > \log_{10}(125000)$ or $\log_{10}(a^b) = b*\log_{10}(a)$
$\Rightarrow n-1 > \log_{10}(125000)/\log_{10}(2)$
$\Rightarrow n > (\log_{10}(125000)/\log_{10}(2)) + 1 \Rightarrow n > 17{,}93 \Rightarrow$ **n=18**

Vérification : pour n = 18, $N_{10} \in [-2^{17}, 2^{17}-1]$ soit $\mathbf{-131072 \leq N_{10} \leq 131071}$

4. Représentation approximative des nombres réels

4.1 Nombres décimaux

L'encodage des nombres ***flottants*** est inspiré de l'écriture scientifique des nombres décimaux qui se compose d'un **signe** (+ ou -), d'un nombre décimal *m* appelé ***mantisse***, compris dans l'intervalle **[1, 10[** et d'un entier relatif *n* appelé exposant. L'écriture scientifique d'un nombre décimal est de la forme $\mathbf{\pm m \times 10^n}$.

> On remarquera que **le nombre 0 ne peut pas être représenté par cette écriture**.

4.2 Norme IEEE 754

Ressource

- IEEE 754 Wikipédia

Présentation

La représentation des nombres *flottants* et les opérations arithmétiques sur ces nombres a été définit par la norme IEEE 754. C'est la norme couramment utilisée dans les ordinateurs. Selon la précision souhaitée, la norme définit un format sur **32 bits** (**simple précision**) et un autre sur **64 bits** (**double précision**).

> Un nombre flottant a la forme suivante : $\mathbf{(-1)^s m \times 2^{(e-d)}}$.
> **s** : signe (0 ⇒ positif, 1 ⇒ négatif)
> **m** : mantisse comprise dans l'intervalle [1,2[
> **e** : exposant, décalé (**biaisé**) d'une valeur **d** qui dépend du format choisi (32 ou 64bits)

Remarques

- La **mantisse** est comprise dans l'intervalle [1,2[donc *m* est toujours de la forme 1,xxxx. Aussi pour gagner 1 bit, **on ne stocke que les chiffres après la virgule**, qu'on appelle la ***fraction***.
- L'**exposant** peut être positif ou négatif. Cependant, la représentation habituelle des nombres signés (**complément à 2**) **n'est pas utilisée** car, elle rendrait la comparaison entre les nombres flottants un peu plus difficile. Pour régler ce problème, l'exposant est « **biaisé** », afin de le **stocker sous forme d'un nombre non signé**.

Format simple précision (32 bits)

Un nombre flottant simple précision est stocké dans un mot de 32 bits : **1 bit de signe**, **8 bits pour l'exposant** et **23 bits pour la mantisse**.

Pour le format **32 bits**, l'**exposant décalé (e-d)** est un entier sur **8bits** qui représente les nombres entre 0 et 255. **d=127**, on représente ainsi des exposants signés dans l'intervalle [-126, 127] car, 0 et 255 représentent des nombres particuliers (voir tableau des valeurs spéciales).

Exemple

Conversion en décimal de $\mathbf{N_2}$ **= 1 10001100 10101101100000000000000**

- signe = -1^1 = **-1**
- exposant = $(2^7 + 2^3 + 2^2) - 127 = (128 + 4 + 2) - 127$ = **7**
- mantisse = $1 + 2^{-1} + 2^{-3} + 2^{-5} + 2^{-6} + 2 + 2^{-8} + 2^{-9}$ = **1,677734375**

donc $\mathbf{N_{10}} = - 1{,}677734375 * 2^7$ = **-214,75**

Tableau récapitulatif simple précision, double précision

	exposant (e)	fraction (f)	valeur
32 *bits*	8 *bits*	32 *bits*	**$(-1)^s$ x 1,f x $2^{(e-127)}$)**
64 *bits*	11 *bits*	52 *bits*	**$(-1)^s$ x 1,f x $2^{(e-1023)}$)**

s : signe (0 ⇒ positif, 1 ⇒ négatif)
f : fraction de la mantisse m
e : exposant

> En **simple précision (32 bits)**, les nombres flottants positifs peuvent représenter les nombres décimaux compris dans l'intervalle **[10^{-38},0^{38}]**, **[10^{-308},0^{308}] en double précision (64 bits)**.

Valeurs spéciales

Le format des nombres flottant ne permettant pas de représenter le nombre 0, la norme IEE 754 utilise les valeurs de l'exposant 0 et 255 pour remédier à ce problème et représenter d'autres valeurs spéciales.

Signe	Exposant	Fraction	Valeur spéciale
0	0	0	+0
1	0	0	-0
0	255	0	+∞
1	255	0	-∞
0	255	≠0	NaN

NaN : Not a Number permet de représenter des résultats d'opérations invalides (0/0 etc.)

Nombres dénormalisés

Dans la représentation précédente (**normalisée**), le plus petit nombre est 2^{-126} soit environ 10^{-38}. Il n'y a **pas de nombre représentable** dans l'intervalle [0, 2^{-126}[.

> La norme **IEEE 754** permet d'encoder des nombres de la forme **$(-1)^s$ x 0,f x 2^{-126}**. Sans les **nombres dénormalisés**, les deux tests x - y = 0 et x = y ne seraient pas systématiquement équivalent.

Arrondis

> La représentation des nombres décimaux par des flottants est une représentation **approximative**. La norme IEEE 754 définit quatre modes d'arrondi pour choisir le meilleur flottant.

- *Au plus près*(par défaut) : le flottant le plus proche de la valeur exacte;
- *Vers zéro* : le flottant le plus proche de 0;
- *Vers plus l'infini* : le plus petit flottant supérieur ou égal à la valeur exacte;
- *Vers moins l'infini* : le plus grand flottant inférieur ou égal à la valeur exacte;

Exemple Le nombre flottant le plus proche de $\mathbf{N_{10} = 1,6}$ est $\mathbf{N_2}$ **= 0 01111111 10011001100110011001101**

$N_{10} = 1 + (2^{-1} + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + 2^{-16} + 2^{-17} + 2^{-20} + 2^{-21} + 2^{-23}) \times + 2^{127-127} =$ 1,600000023841857910156 25

4.3 Les flottants en Python

Voir Variables, types numériques et entrées / sorties dans la console

5. Représentation des caractères

> La représentation des caractères dans les ordinateurs permet de **stocker et échanger** des textes. Associer un numéro unique à un caractère s'appelle l'**encodage**

5.1 Codage ASCII

Dans les années 60, les ordinateurs utilisent des mots de 8 octets. Le code ASCII (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange) normalise la représentation des caractères. Dans ce code, le bit de poids fort $\mathbf{b_7}$ **est toujours à 0**. Entre $\mathbf{00_{16}}$ et $\mathbf{20_{16}}$ les caractères sont dit spéciaux (espace, tabulation, fin de ligne, etc).

USASCII code chart

Bits b_7 b_6 b_5 →	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
b_4 b_3 b_2 b_1 ↓ Row / Column →	0	1	2	3	4	5	6	7
0 0 0 0	0	NUL	DLE	SP	0	@	P	`	p
0 0 0 1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	2	STX	DC2	"	2	B	R	b	r
0 0 1 1	3	ETX	DC3	#	3	C	S	c	s
0 1 0 0	4	EOT	DC4	$	4	D	T	d	t
0 1 0 1	5	ENQ	NAK	%	5	E	U	e	u
0 1 1 0	6	ACK	SYN	&	6	F	V	f	v
0 1 1 1	7	BEL	ETB	'	7	G	W	g	w
1 0 0 0	8	BS	CAN	(	8	H	X	h	x
1 0 0 1	9	HT	EM	)	9	I	Y	i	y
1 0 1 0	10	LF	SUB	*	:	J	Z	j	z
1 0 1 1	11	VT	ESC	+	;	K	[	k	{
1 1 0 0	12	FF	FS	,	<	L	\	l	|
1 1 0 1	13	CR	GS	-	=	M	]	m	}
1 1 1 0	14	SO	RS	.	>	N	^	n	~
1 1 1 1	15	SI	US	/	?	O	_	o	DEL

Un texte codé en ASCII est simplement une suite d'octets correspondant à la séquence de caractères.

```
 C  e  c  i     e  s  t     u  n     t  e  x  t  e  !
43 65 63 69 20 65 73 74 20 75 6e 20 74 65 78 74 65 21
```

Le code ASCII est encore utilisé aujourd'hui. Cependant, il a l'inconvénient de ne pas représenter les caractères accentués. Il est également limité à l'alphabet latin.

> De nombreux autres encodages ont été proposés et utilisés pour résoudre ces problèmes. Le code qui a fini par s'imposer au niveau international est le code **UTF-8** (**U**nicode **T**ransformation **F**ormat).

5.1 ISO 8859

- **Ressource** : ISO-8859-1 sur Wikipédia

A compléter

> Bien que les pages ISO-8859-n permettent l'encodage d'un très grand nombre de caractères, elles **ne conviennent pas** quand on souhaite écrire un texte avec un **mélange de caractères** dans différentes pages.

5.3 Codage Unicode

La norme Unicode, développée par un consortium du même nom, définit un jeu de caractères qui vise à inclure le plus grand nombre possible de systèmes d'écriture. Il contient à l'heure actuelle (2021) 143000 caractères couvrant 150 systèmes d'écriture ainsi que les emojis.

> Chaque caractère a un code unique, appelé "*code point*" ou **point de code** et noté **U+XXXX** où chaque X est un caractère hexadécimal. En pratique la séquence de X est au minimum de 4 caractères.

Cette norme définit plusieurs techniques d'encodage pour représenter les points de code de manière plus ou moins économique selon la technique choisie. Ces encodages, appelés *formats de transformation universelle* ou ***U**niversal **T**ransformation **F**ormat* (**UTF**) en anglais, portent les noms **UTF-n**, où *n* indique le nombre minimal de *bits* pour représenter un point de code.

UTF-8

5.3.1 UTF-8

> UTF-8 (**U**niversal **T**ransformation **F**ormat) est un **code à taille variable** destiné à représenter les caractères Unicode avec au minimum **8 bits**. C'est le format le plus utilisé sous Linux, dans les protocoles réseaux et les sites Web.

Les caractères sont représentés sur **1,2,3 ou 4 octets**. UTF-8 est **compatible** avec le code **ASCII** : les codes UTF-8 d'un octet avec le bit de poids fort à zéro sont les caractères du code ASCII (ainsi les programmes qui fonctionnaient sur des textes encodés en ASCII devraient continuer à fonctionner si ces mêmes textes sont encodés en UTF-8.
Les caractères codés sur plus d'un octet ont tous le bit de poids fort à 1, de telle sorte qu'ils sont en général ignorés par les logiciels qui ne connaissent que le code ASCII.

Nb Octets	Premier "code point"	Dernier "code point"	Octet 1	Octet 2	Octet 3	Octet 4
1	U+0000	U+007F	0xxxxxxx			
2	U+0080	U+07FF	110xxxxx	10xxxxxx		

Nb Octets	Premier “code point”	Dernier “code point”	Octet 1	Octet 2	Octet 3	Octet 4
3	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Principe de l'encodage

- Si le bit de poids fort d'un octet est à 0, alors il s'agit d'un caractère ASCII codé sur les 7 bits restants.
- Sinon, les premiers bits de poids fort de l'octet indiquent le nombre d'octets utilisés pour encoder le caractère à l'aide d'une séquence de bits à 1 et se termine par un bit à 0. Exemple : 110xxxxx signifie que le caractère est codé sur 2 octets.
- Dans le cas d'un encodage sur k octets, les k-1 octets qui suivent l'octet de poids fort doivent tous être de la forme 10xxxxxx.

Exemples

Pt de code	point de code binaire	UTF-8 (binaire/hexa)
U+004B	01001011	01001011 (4b)
U+00C5	11000101	11000011 (C3) 10000101 (85)
U+0A9C	00001010 10011100	11100000 (E0) 10101010 (AA) 10011100 (9C)

5.3.2 UTF16

Voir Unicode sur Wikipédia.

5.3.3 UTF-32

Voir Unicode sur Wikipédia.

5.3.4 UNICODE et Python

> Les chaînes de caractères en Python sont des séquences de caractères au format **UTF-8**. Ces caractères peuvent être saisi directement avec leur point de code à l'aide d'une séquence d'échapement '\u' suivie du point de code en hexadécimal.

Exemple

*.py

```
a = '\u0042'
```

```
print(a) # Affichage du caractère A
```

A compléter

5.4 Les chaînes de caractères en Python

Voir Les séquences : chaînes de caractères

Résumé

- Les **nombres entiers** sont représentés en **binaire** (base 2) dans un ordinateur. La notation **hexadécimale** (base 16) simplifie l'écriture des nombres binaires. Il existe deux représentations des nombres stockés sur plusieurs octets, **le petit et le gros boutisme**, qui se différencient selon l'ordre des octets en mémoire. Le codage des nombres relatifs utilise la technique du **complément à deux** pour faciliter les opérations arithmétiques.
- Les **nombres flottants** sont une représentation **approximative** des nombres réels dans un ordinateur. La norme internationale IEEE 754 définit un encodage en simple (32 bits) ou double précision (64 bits) ainsi que des règles d'**arrondi**. Les opérations arithmétiques sur les nombres flottants n'ont pas toujours les mêmes propriétés que ces mêmes opérations sur les réels.
- Les caractères sont représentés par des entiers et des tables de correspondances sont établies par des normes (ASCII, Unicode). La norme ASCII définit un jeu restreint de 127 caractères. Le format Unicode est conçu pour représenter les caractères de toutes les langues. Il existe plusieurs formats d'encodage Unicode (UTF-8, UTF-16, UTF-32) qui se distinguent par le nombre d'octets minimum pour représenter les points de code.