1ère NSI — Thème 4: Représentation des données

Types & Valeurs de base

Lycée Fustel de Coulanges, Massy

Marc Biver, novembre 2023, *v0.2*

Ce document reprend les notions abordées en cours et pratiquées en TP / TD; il inclut la totalité de ce qui a été diffusé en classe sur ce thème. Si vous avez des questions à son propos n'hésitez pas à me contacter par le biais de la messagerie de l'ENT.

Table des matières

1 Point d'étape – où est-on / où va-t-on ?

1.1 Ce qu'on a couvert jusqu'à présent

- Rudiments de l'architecture physique d'un ordinateur - le modèle de Von Neumann.
- Mise en jambes sur de l'écriture de code : réalisation d'une page Web en HTML.
- Et surtout : introduction à Python, et plus spécifiquement :
  - Ce qu'on appelle ses "constructions élémentaires" – variables, fonctions, conditions & embranchements, boucles...
  - Les types et valeurs de base : entiers (naturels et relatifs), flottants (réels), chaînes de caractères et booléens (qu'on n'a que brièvement abordés pour l'instant).
  - Un type dit "construit" – les listes.

1.2 Ce dont on va parler dans ce nouveau chapitre

On va prendre un pas de recul par rapport au chapitre sur Python – et se demander comment les données qu'on manipule sont stockés dans l'ordinateur. Après tout, on sait que celui-ci ne manipule que des 0 et des 1 ; donc comment peut-il stocker `12` ou π ou `"Bonjour!!"` ?

C'est à ces questions qu'on va répondre en étudiant comment différents types de données sont ***représentés*** dans la mémoire de l'ordinateur

Spécifiquement, on va aborder les points suivants :

- Représentation des entiers naturels ;
- Représentation des entiers relatifs ;
- Représentation des nombres réels ("nombres flottants") ;
- Représentation et codage du texte ;
- Représentation des booléens – opérateurs booléens, logique boolénne – ce qui nous amènera à :
- Circuits logiques élémentaires.

1.3 Comment on va procéder

On passe ici dans un mode beaucoup plus théorique :

- On ne cherche plus à *faire faire* quelque chose à l'ordinateur (comme en programmation) ;
- On cherche plutôt à comprendre *comment l'ordinateur lui-même fonctionne.*
- On va s'intéresser à des aspects beaucoup plus mathématiques de l'informatique.

On va donc devoir travailler différemment du chapitre sur l'introduction à Python :

- **Prise de notes essentielle** ;
- Plusieurs exercices sur papier qu'on fera en classe et dont il sera très important que vous gardiez une trace ;

— <u>**Attention :**</u> régulièrement je vous demanderai de terminer à la maison les exercices commencés en classe – il faudra donc en avoir pris suffisamment de notes !
— Quand même, quelques applications / exercices sur machine.

2 Bases de numération & Représentation des entiers naturels

2.1 La base 10

Quelques questions "simples" pour commencer...

→*Que veut dire "2754"?*

> *Implicitement, c'est une somme de puissances de 10 :*
>
> $$2 \cdot 10^3 + 7 \cdot 10^2 + 5 \cdot 10^1 + 4 \cdot 10^0$$

→*Autrement dit : comment fonctionne la base 10?*

> *Plus généralement, c'est donc une somme de puissances de 10 lue de droite à gauche (puisque si l'on ne regarde que le premier chiffre d'un nombre on ne peut pas savoir à quelle puissance il correspond). Cela revient à dire qu'un nombre* `ABCDEF` *écrit en base 10 se lit :*
>
> $$F \cdot 10^0 + E \cdot 10^1 + D \cdot 10^2 + C \cdot 10^3 + B \cdot 10^4 + A \cdot 10^5$$

→*Pourquoi la base 10 s'appelle-t-elle la base 10?*

> *Tout simplement parce qu'elle permet d'écrire la totalité de tous les entiers naturels en n'utilisant que 10 symboles – les 10 chiffres arabes : 1, 2, 3, 4, 5, 6, 7, 8, 9 – et le plus important de tous : 0.*

→*Pourquoi utilise-t-on la base 10?*

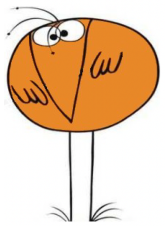
> *Deux remarques pour commencer : d'abord, on n'utilse pas que la base 10 – pensez par exemple aux oeufs qu'on compte encore aujourd'hui par douzaines ; ensuite, il n'y a pas de réponse certaine à cette question. La plus intuitive est que cela correspond à nos doigts – on en a 10, donc quand on les a tous "utilisés" on repart du premier, comme on fait avec les chiffres à l'écrit.*

→*Comment s'appellent les symboles utilisés pour écrire des nombres en base 10?*

> *Ca a été dit plus haut – ce sont les chiffres, inventés en Inde, puis parvenus en Europe par le monde arabe (d'où leur nom) aux environs du Xe siècle.*

2.2 Et en base ~~Shadok~~ 4, par exemple ?

Les Shadoks, c'est un dessin animé absurde français des années 60 – l'histoire d'oiseaux aux ailes ridiculement petites habitant sur une autre planète qui veulent conquérir l'univers ; pour ça ils ont besoin de la science. Donc de pouvoir compter. Leur problème ? Leur langue ne contient en tout et pour tout que quatre mots : `GA`, `BU`, `ZO`, et `MEU`....



Alors comment faire.... ?

Allons voir la solution qu'ils ont retenue...

Récapitulons :

Mot	Symbole	Quantité
GA		-
BU		I
ZO		I I
MEU		I I I

Exercice 1: conversions d'une base à une autre

Convertissez en base 10 les quantités exprimées en base Shadok suivantes :

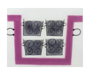
a. Zo

b. Bu Zo

c. Meu Meu

d. Zo Ga

e. Meu Zo Ga

Souvenez-vous : on compte, de droite à gauche, des Shadoks, puis des "poubelles"  *de quatre Shadoks, des "grandes poubelles"*  *qui contiennent chacune quatre poubelles, etc...*

Prenons d'entrée de jeu ici l'habitude d'aborder les nombres de droite à gauche – donc en commençant par les Shadoks, puis les poubelles, puis les grandes poubelles, etc...

a. *Zo = 2 – c'est écrit dans le tableau.*

b. *Bu Zo – on a zo, donc deux, Shadoks (chiffre de droite) ; et on a bu, donc une, poubelle qui contient 4 Shadoks. Donc Bu Zo* $= 2 \times 1 + 1 \times 4 = 6$.

c. *Meu Meu – même raisonnement : meu Shadoks, donc 3, et meu poubelles, donc* 3×4 *; donc Meu Meu* $= 3 \times 1 + 3 \times 4 = 15$.

d. *Zo Ga – même chose, sauf que cette fois il y a ga, donc zéro, Shadoks a droite : Zo Ga* $= 0 \times 1 + 2 \times 4$

e. *Meu Zo Ga – toujours la même démarche, sauf qu'on a cette fois un chiffre de plus, donc des grandes poubelles contenant chacune 4 poubelles de 4 :*

$$0 \times 1 + 2 \times 4 + 3 \times 16 = 56$$

Exercice 2: et maintenant sauvons le pauvre Professeur Shadoko...

Convertissez le nombre mystère 

en français.

Même chose que dans l'exercice précédent, sauf qu'on introduit ici un quatrième chiffre – les "super poubelles" qui contiennent 4 grandes poubelles, qui contiennent 4 poubelles, qui contiennent 4 Shadoks – donc $4 \times 4 \times 4$; il serait peut-être temps de parler en puissances de 4, non ?

$$Meu \times 4^0 + Ga \times 4^1 + Zo \times 4^2 + Bu \times 4^3$$

$$= 3 \times 1 + 0 \times 4 + 2 \times 16 + 1 \times 64 = 99$$

Exercice 3: plus généralement...

Proposez une méthode pour convertir un nombre Shadok dans notre système d'écriture courant.

On ne propose ici qu'une généralisation de ce qu'on a fait jusqu'à present : partir du chiffre de droite, le multiplier par 4^0, soit 1 ; passer au chiffre suivant, le multiplier par 4^1, soit 4, et l'ajouter au précédent ; continuer ainsi vers la gauche en augmentant à chaque fois les puissances de 4 et en ajoutant au precedent jusqu'à atteindre le chiffre le plus à gauche.

En pseudo-code, on pourrait écrire :

```
function ConvertirBase4enBase10(nombreBase4)
    résultat ← 0
    puissance ← 0
    for all chiffre de nombreBase4, de droite à gauche do
                ▷ On parcourt tous les chiffres constitutifs du nombre en entrée
        résultat ← résultat + chiffre×4^puissance
        puissance ← puissance + 1
    end for
    return résultat
end function
```

Bon – abandonnons le vocabulaire Shadok à présent et revenons au cours : on parle évidemment ici d'une **base 4** et notre méthode générale peut s'écrire sous la forme d'une suite de puissances de 4.

Pour un nombre N noté en base 4 $a_4a_3a_2a_1a_0$, par exemple, on le convertira par la formule :

$$N = a_0 \cdot 4^0 + a_1 \cdot 4^1 + a_2 \cdot 4^2 + a_3 \cdot 4^3 + a_4 \cdot 4^4$$

Ou encore :

$$N = a_0 \cdot 1 + a_1 \cdot 4 + a_2 \cdot 16 + a_3 \cdot 64 + a_4 \cdot 256$$

Et plus généralement, pour un nombre N noté en base 4 $a_ka_{k-1}(...)a_2a_1a_0$, on aura :

$$N = \underbrace{a_0 \cdot 4^0 + a_1 \cdot 4^1 + a_2 \cdot 4^2 + (...) + a_{k-1} \cdot 4^{k-1} + a_k \cdot 4^k}_{\text{(k+1) éléments, de 0 à k}}$$

ce qui s'écrit aussi :

$$N = \sum_{i=0}^{k} a_i \cdot 4^i$$

Donc si on remplace Ga, Bu, Zo, et Meu par les chiffres arabes correspondants (0, 1, 2, 3), on pourra dire que le nombre 132 en base 4 vaut 46 en base 10 :

$$2 \cdot 4^0 + 3 \cdot 4^1 + 1 \cdot 4^2 = 2 \cdot 1 + 3 \cdot 4 + 2 \cdot 16 = 46$$

Pour exprimer cela simplement, on va placer la base d'expression du nombre en indice de celui-ci et écrire : $132_4 = 46_{10}$

Exercice 4: et dans l'autre sens ?

Convertissez en base 4 (ou en Shadok, comme vous préférez) les nombres écrits en base 10 suivants :

a. $16_{10} =$
b. $17_{10} =$
c. $64_{10} =$
d. $65_{10} =$
e. $11_{10} =$
f. $25_{10} =$
g. $57_{10} =$
h. $675_{10} =$

On reprend à partir de maintenant les chiffres arabes pour noter nos nombres – ceux de 0 à 9 quand on est en base 10, et ceux de 0 à 3 en base 4 (qui remplacent donc Ga a Meu en base Shadok).

On a remarqué en classe que les 4 premiers exemples de cet exercice sont des valeurs ***très*** *proches de puissances de 4 – et ce n'est évidemment pas un hasard :*

$$\begin{aligned} 16_{10} &= (4^2)_{10} \\ &= (1 \times 4^2 + 0 \times 4^1 + 0 \times 4^0)_{10} \\ &= (100)_4 \end{aligned}$$

$$\begin{aligned} 17_{10} &= (16 + 1)_{10} \\ &= (4^2 + 1)_{10} \\ &= (1 \times 4^2 + 0 \times 4^1 + 1 \times 4^0)_{10} \\ &= (101)_4 \end{aligned}$$

Et de la même manière, on obtient pour les deux suivants :

$$64_{10} = (4^3)_{10} = 1000_4$$

$65_{10} = (4^3 + 1)_{10} = 1001_4$

Pour les questions e) à h), on applique rigoureusement la méthode que l'on va decrire a l'exercice suivant et on obtient :

e. $11_{10} = 23_4$
f. $25_{10} = 121_4$
g. $57_{10} = 321_4$
h. $675_{10} = 22203_4$

Exercice 5: ça commence à devenir un poil compliqué...

Proposez cette fois une méthode pour convertir un nombre de base 10 en base 4 (ou Shadok).

Pour convertir le nombre N_{10} en base 4, on va procéder de la manière qui suit :

1. *On cherche d'abord la première puissance k de 4 qui soit strictement supérieure à N_{10}*
2. *On dresse un tableau avec les premières puissances de 4, de 4^{k-1} à 4^0.*
3. *On effectue une division euclidienne de N_{10} par 4^{k-1} et on note le quotient de cette division dans la case 4^{k-1} du tableau.*
4. *On soustrait 4^{k-1l} de N_{10}a plus grande puissance de 2 qui est inférieure au nombre à convertir.*
5. *On prend le reste de cette division et on répète ces étapes jusqu'à arriver à un quotient inférieur à 4.*

Exemple, pour convertir le nombre 710_{10} :

$256 \leq 710 < 1024$

Donc : $4^4 \leq 710 < 4^5$

Donc on dresse le tableau des puissances de 4 de l'exposant 4 à l'exposant 0 :

4^4	4^3	4^2	4^1	4^0
256	64	16	4	1
2	3	0	1	2

$710//256$	$= 2$	*(reste = 198)*	*(on place un "2" dans la case 256)*
$198//64$	$= 3$	*(reste = 6)*	*(on place un "3" dans la case 64)*
$6//4$	$= 1$	*(reste = 2)*	*(on place un "1" dans la case 4)*
$2 < 4$			*(on place un "2" dans la case 1)*
Fin			

Le nombre 710_{10} se convertit donc en 23012_4.

2.3 La base 2 – ou "numération binaire"

Quel rapport avec l'informatique ?

Imaginons un interrupteur électrique : il peut être soit en position *ON*, soit en position *OFF*. En informatique, le "coeur" d'un ordinateur est composé de millions voire de milliards de petits "interrupteurs" appelés ***transistors***.

Un transistor permet à un courant électrique de passer (état "allumé" – que l'on va appeler "1") ou non (état "éteint" ou "0"). C'est cette simplicité qui rend les transistors fiables et efficaces pour construire des circuits électroniques complexes comme ceux que l'on trouve dans les processeurs d'ordinateur. Ainsi, la mémoire de l'ordinateur utilise des transistors pour stocker des informations sous forme de 1 et de 0 selon l'état des transistors qui la composent .

Toute information, de quelque nature qu'elle soit, doit être convertie en 1 et en 0 pour être manipulée par un ordinateur.

Donc en base 2.

Donc en binaire.

Conversions de et vers la base 2

C'est à présent que les Shadoks vont venir à notre secours : parce que le principe en base 2 est rigoureusement le même qu'en base 4 – sauf qu'au lieu d'utiliser 4 symboles (0, 1, 2, et 3), on n'en utilise que 2 (0 et 1) et qu'au lieu de manipuler des puissances de 4, on manipule des puissances de 2. Pour un nombre N noté en base 2 $a_k a_{k-1}(...)a_2 a_1 a_0$, on aura :

$$N = \underbrace{a_0 \cdot 2^0 + a_1 \cdot 2^1 + a_2 \cdot 2^2 + (...) + a_{k-1} \cdot 2^{k-1} + a_k \cdot 2^k}_{\text{(k+1) éléments, de 0 à k}}$$

ce qui s'écrit aussi :

$$N = \sum_{i=0}^{k} a_i \cdot 2^i$$

Conseil:

Vous allez vite vous rendre compte que pour ce chapitre et pour la matière NSI en général on gagne beaucoup de temps en connaissant les premières puissances de 2 – je vous conseille donc vivement de faire l'effort de les retenir :

$2^{10} = 1024$	$2^9 = 512$	$2^8 = 256$	$2^7 = 128$	$2^6 = 64$	$2^5 = 32$
$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$	

Exercice 6: commençons par compter...

Comptez de 1 en 1 en base 2 en partant de 0 et jusqu'à arriver au nombre qui, en base 10, est représenté par 17.
Souvenez-vous : pour compter en base 4, on faisait Ga, Bu, Zo, Meu, Bu Ga, Bu Bu, Bu Zo... Soit 0, 1, 2, 3, 10, 11, 12... En base 2, c'est la même chose, sauf qu'on n'a que deux symboles, 0 et 1.

0 ; 1 ; 10 ; 11 ; 100 ; 101 ; 110 ; 111 ; 1000 ; 1001 ; 1010 ; 1011 ; 1100 ; 1101 ; 1110 ; 1111 ; 10000 ; 10001

Vous saviez que l'humanité se divisait en 10 parties ? Ceux qui savent compter *en binaire et ceux qui ne savent pas...*

Exercice 7: et maintenant convertissons de base 2 en base 10

En utilisant les techniques utilisées pour la base 4, convertissez les nombres binaires suivants en base 10 :

a. 11

b. 10110

c. 10101100

On va faire ici une somme des puissances de 2 successives, comme on l'avait fait avec les puissances de 4 plus haut dans le cours. Je vous soumets ici les réponses en détaillant les calculs uniquement pour le dernier exercice :

a. $11_2 = 3_{10}$

b. $10110_2 = 22_{10}$

Convertissons à présent 10101100_2 *en parcourant ses chiffres de droite à gauche en incrémentant à chaque étape la puissance appliquée à 2, et en incrémentant à chaque fois le résultat (que nous initialisons a 0) :*

— $resultat += 0 \times 2^0 = 0$
— $resultat += 0 \times 2^1 = 0$
— $resultat += 1 \times 2^2 = 4$
— $resultat += 1 \times 2^3 = 12$
— $resultat += 0 \times 2^4 = 0$
— $resultat += 1 \times 2^5 = 44$
— $resultat += 0 \times 2^6 = 0$
— $resultat += 1 \times 2^7 = 172$

Donc $10101100_2 = 172_{10}$

Passons maintenant à la conversion réciproque – de base 10 vers base 2. Il existe en fait deux méthodes possibles :

— Celle qu'on a vue en classe pour la base 4, qui s'appelle pour la base 2 "**méthode des soustractions successives**" (*parce qu'en base 2 on n'a pas besoin de diviser !*) :

1. On commence par dresser un tableau avec les premières puissances de 2.
2. On soustrait la plus grande puissance de 2 qui est inférieure au nombre à convertir.
3. On note un "1" dans la case correspondante du tableau.
4. On répète ces étapes pour le nouveau nombre à convertir jusqu'à arriver à 0.

Exemple, pour convertir le nombre 25_{10} :

32	16	8	4	2	1
0	1	1	0	0	1

$$\begin{array}{lll} 25-16 & =9 & \text{(on place un "1" dans la case 16)} \\ 9-8 & =1 & \text{(on place un "1" dans la case 8)} \\ 1-1 & =0 & \text{(on place un "1" dans la case 1)} \\ \text{Fin} & & \end{array}$$

Le nombre 25_{10} se convertit donc en 11001_2.

— Celle des "**divisions successives**" qui consiste à diviser le nombre donné par 2 de manière répétée, et à noter le reste à chaque étape. Le processus continue en utilisant le quotient obtenu comme nouveau nombre à diviser, jusqu'à ce que le quotient soit égal à 0.

Exemple : convertissons de nouveau, mais avec cette nouvelle méthode, 25_{10} en base 2.

$$\begin{array}{ll} 25/2=12 & \text{reste } 1 \\ 12/2=6 & \text{reste } 0 \\ 6/2=3 & \text{reste } 0 \\ 3/2=1 & \text{reste } 1 \\ 1/2=0 & \text{reste } 1 \end{array}$$

En lisant les restes à l'envers, on obtient 11001_2.

Exercice 8: mettons ces deux méthodes en pratique

En utilisant les deux méthodes, convertissez en base 2 :

a. 13
b. 22
c. 128

1$^{\text{ère}}$ méthode (soustractions successives) :

a. 13 :

16	8	4	2	1
0	1	1	0	1

$$\begin{array}{lll} 13-8 & =5 & \textit{(on place un "1" dans la case 8)} \\ 5-4 & =0 & \textit{(on place un "1" dans la case 4)} \\ 1-1 & =0 & \textit{(on place un "1" dans la case 1)} \\ \textit{Fin} & & \end{array}$$

Le nombre 13_{10} se convertit donc en 1101_2.

b. 22 :

32	16	8	4	2	1
0	1	0	1	1	0

$$\begin{array}{lll} 22-16 & =6 & \textit{(on place un "1" dans la case 16)} \\ 6-4 & =2 & \textit{(on place un "1" dans la case 4)} \\ 2-2 & =0 & \textit{(on place un "1" dans la case 2)} \\ \textit{Fin} & & \end{array}$$

Le nombre 22_{10} se convertit donc en 10110_2.

c. 128 :

256	128	64	32	16	8	4	2	1
0	1	0	0	0	0	0	0	0

$128 - 128 \quad = 0$ *(on place un "1" dans la case 128)*
Fin

Le nombre 128_{10} se convertit donc en 1000000_2.

2^nde^ méthode (divisions successives) :

a. 13 :

$$\begin{array}{ll} 13/2 = 6 & \textit{reste } 1 \\ 6/2 = 3 & \textit{reste } 0 \\ 3/2 = 1 & \textit{reste } 1 \\ 1/2 = 0 & \textit{reste } 1 \end{array}$$

En lisant les restes à l'envers, on obtient bien 1101_2.

b. 22 :

$$\begin{array}{ll} 22/2 = 11 & \textit{reste } 0 \\ 11/2 = 5 & \textit{reste } 1 \\ 5/2 = 2 & \textit{reste } 1 \\ 2/2 = 1 & \textit{reste } 0 \\ 1/2 = 0 & \textit{reste } 1 \end{array}$$

En lisant les restes à l'envers, on obtient bien 10110_2.

c. 128 :

$$\begin{array}{ll} 128/2 = 64 & \textit{reste } 0 \\ 64/2 = 32 & \textit{reste } 0 \\ 32/2 = 16 & \textit{reste } 0 \\ 16/2 = 8 & \textit{reste } 0 \\ 8/2 = 4 & \textit{reste } 0 \\ 4/2 = 2 & \textit{reste } 0 \\ 2/2 = 1 & \textit{reste } 0 \\ 1/2 = 0 & \textit{reste } 1 \end{array}$$

En lisant les restes à l'envers, on obtient bien 10000000_2.

Astuce:

Il est essentiel de savoir effectuer ces conversions "à la main" comme on vient de le faire – mais pour de grands nombres, ça prend du temps, et Python vous offre une solution beaucoup plus rapide :

```python
# Convertir '10110 ' en base 10
nombre_decimal = int('10110 ', base =2)
print ( nombre_decimal ) # Output sera 22
```

Donc ca, c'est comment l'ordinateur stocke et manipule les nombres – c'est pratique pour lui, mais un peu complique pour nous : par exemple, la population de la France est d'environ 64,8 millions de personnes. En binaire ca nous donne :

1111011100110001010000000... Même celle de Massy, qui n'est que de 50.506 habitants, donne 1100010101001010. Vous trouvez ca lisible, vous ?

2.4 La base hexadécimale, ou base 16

Cette base permet un compromis entre le code binaire des machines et une base de numération pratique à utiliser pour les humains. Elle utilise 16 symboles, ou "chiffres" de base : les dix chiffres arabes et les lettres A, B, C, D, E, F. Compter de 0_{10} a 20_{10} dans différentes bases se fait de la manière suivante :

Base 10	**Base 2**	**Base 16**
01	0001	1
02	0010	2
03	0011	3
04	0100	4
05	0101	5
06	0110	6
07	0111	7
08	1000	8
09	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F
16	10000	10
17	10001	11
18	10010	12
19	10011	13
20	10100	14

Quel sont les intérêts de cette base ?

— Compacité : contrairement à la base 2, on peut écrire de grosses quantités en utilisant peu de symboles – la population de la France, c'est 3DCC500 et celle de Massy C54A.

— La conversion de base 2 à base 16 est particulièrement simple, puisque $16 = 2^4$ – on va y revenir ci-dessous.

La conséquence de ces points est que, vous allez le constater, la base hexadécimale est omniprésente en informatique – ce qui explique pourquoi on l'étudie ici : on la retrouve dans l'adressage en mémoire, dans les adresses réseau, dans la cryptographie, dans le codage des couleurs en programmation web...

Rapport a la base 2

Expliquons un peu plus l'intérêt qu'il y a à utiliser une base qui soit aussi une puissance de 2 : comme vous le voyez sur le tableau ci-dessus, les 16 premiers entiers correspondent exactement aux représentations sur 4 symboles : de 0000 (soit 0 en base 10) à 1111 (soit 15 en base 10). La conversion se fait donc de manière très aisée :

— De base 2 en base 16 : on découpe, en partant de la droite, le nombre en "tranches" de 4 symboles et on convertit chacune de ces tranches en son symbole hexadécimal correspondant.
— De base 16 en base 2 : on fait exactement l'inverse – on convertit un par un les symboles en leur équivalent binaire.

Exercice 9: pratiquons ca un peu...

Convertissez en base 2 ou 16 les quantités suivantes (utilisez le tableau ci-dessus !) :

a. $3E6_{16}$
b. $A420_{16}$
c. 110100111100_2
d. 1101001111001_2

a. $3E6_{16}$ *: on s'appuie sur le tableau précédent et on traduit symbole par symbole en binaire :* $3_{16} = 0011_2$ *;* $E_{16} = 1110_2$ *;* $6_{16} = 0110_2$*. Donc en éliminant les 0 sur la gauche cela donne :* $3E6_{16} = 1111100110_2$
b. $A420_{16} = 1010010000100000_2$
c. 110100111100_2 *: toujours en s'appuyant sur le tableau précédent, on effectue l'opération inverse en découpant le nombre binaire en "tranches" de 4 chiffres, en partant de la droite :*

$$\underbrace{1101}\underbrace{0011}\underbrace{1100}$$

Puis on convertit un à un ces nombres pour obtenir : $D3C_{16}$
d. 1101001111001_2 *– même méthode :*

$$\underbrace{\ 1\ }\underbrace{1010}\underbrace{0111}\underbrace{1001}$$

Et on obtient : $1A79_{16}$

Conversion de & vers la base 10

De nouveau, les principes et méthodes sont exactement les mêmes que pour les bases 4 et 2 – sauf que l'on manipule cette fois 16 symboles et des puissances de 16.

Exercice 10: puisqu'on connait la méthode allons-y directement !

a. Convertissez $12AF_{16}$ en base 10 (en vous souvenant qu'on manipule ici des puissances de 16 : 16, 256, 4096...).
b. Convertissez $A42_{16}$ en base 10.
c. Convertissez 120_{10} en hexadécimal – le plus simple est d'utiliser le méthode des divisions successives, cette fois en divisant par 16 a chaque étape.
d. Convertissez 443_{10} en hexadécimal.

a. $12AF_{16} = (15 \times 16^0 + 10 \times 16^1 + 2 \times 16^2 + 1 \times 16^3)_{10} = 4783_{10}$
b. $A42_{16} = (2 \times 16^0 + 4 \times 16^1 + 10 \times 16^2)_{10} = 2626_{10}$

c. 120_{10} :

$$\begin{array}{ll} 120/16 = 7 & \textit{reste } 8 \\ 7/16 = 0 & \textit{reste } 7 \end{array}$$

En lisant les restes à l'envers, on obtient 78_{16}.

d. 443_{10} :

$$\begin{array}{ll} 443/16 = 27 & \textit{reste } 11 \\ 27/16 = 1 & \textit{reste } 11 \\ 1/16 = 0 & \textit{reste } 1 \end{array}$$

En lisant les restes à l'envers (et en convertissant le nombre décimal 11 en son équivalent hexadécimal soit le chiffre B), on obtient $1BB_{16}$.

Exercice 11: à faire à la maison pour le prochain cours

a. $ABCD_{16}$ en base 10 ;

b. 1896_{16} en base 10 ;

c. 4097_{10} en hexadécimal ;

d. 2023_{10} en hexadécimal.

On applique les mêmes méthodes que précédemment et on obtient :

a. $ABCD_{16} = 43981_{10}$

b. $1896_{16} = 6294_{10}$

c. $4097_{10} = 1001_{16}$

d. $2023_{10} = 7E7_{16}$

2.5 Conversions d'une base à une autre en Python

En Python, par défaut, le nombres affichés ou saisis le sont en base 10 ; mais Python offre différentes fonctions permettant de naviguer aisément entre les bases 2, 10, et 16. Voici quelques exemples saisis en console Python (le préfixe "$>>>$" étant l'invite de commande) :

```
>>> # Utilisation de la notation 0b pour
>>> # manipuler une sequence de bits
>>> 0b100
4
>>> 0b01001011
75

>>> # Dans le sens inverse, on peut utiliser la syntaxe vue plus haut,
>>> # ou bien la fonction bin()
>>> bin(4)
'0b100'
>>> bin(75)
'0b1001011'

>>> # Utilisation de la notation 0x pour manipuler
>>> # des nombres en hexadécimal
>>> 0x10
16
```

```
>>> 0xA0
160
>>> 0xABC
2748

>>> # Et enfin utilisation de la fonction hex()
>>> # pour convertir un décimal en hexadécimal
>>> hex(16)
'0x10'
>>> hex(160)
'0xa0'
>>> hex(2748)
'0xabc'
```

3 Mémoire & encodage des entiers naturels

3.1 Unités de mémoire

En informatique, **TOUT** repose sur des 0 et des 1 – ces chiffres binaires également appelés bit (qui vient de **BI**nary digi**T** en anglais). Alors justement, commençons par un peu de vocabulaire :

Bit : la base de la base, un transistor tout seul qui ne peut avoir que deux valeurs en tout et pour tout : 0 ou 1.

Octet : un groupement de 8 bits – soit 2×4 chiffres binaires que l'on peut donc très simplement écrire sous forme de deux symboles hexadécimaux. C'est pour ca que l'octet est l'unité de mémoire la plus universellement utilisée.

Byte : là, la langue anglaise nous tend un piège. Un byte (prononcé baille-te), c'est un octet ; ***PAS*** un bit.

Kilooctet – ko (kb en anglais) : $10^3 = 1.000$ octets.

Megaoctet – Mo : $10^6 = 1.000.000$ octets.

Gigaoctet – Go : $10^9 = 1.000.000.000$ octets.

Teraoctet – To : $10^12 = 1.000.000.000.000$ octets.

3.2 Les entiers naturels

Encodage / Représentation

On rappelle que l'ensemble des entiers naturels, $\mathbb{N}$, est l'ensemble des nombres entiers positifs, donc compris entre 0 et $+\infty$. Ce sont donc des entiers **non signés**.

Si on veut représenter un tel nombre sur une plage définie de bits, on sera limités dans la plage de valeurs qu'on peut stocker. Par exemple, sur 1 bit on ne peut aller que de 0 a 1 ; sur 4 bits, de 0000 a 1111, soit en décimal de 0 a 15. C'est évidemment peu pratique – et donc en général, on code les entiers plutôt sur 4 *octets*, soit $4 \times 8 = 32$ bits.

Exercice 12: entier maximal sur 1 ou 4 octets

Quel est l'entier maximal que l'on peut coder sur 1 octet (8 bits) ? Et sur 4 octets (32 bits) ?

Sur 8 bits, le nombre maximal que l'on peut coder est $(11111111)_2$ *soit, en décimal :* $2^0 + 2^1 + 2^2 + 2^4 + 2^5 + 2^6 + 2^7 = 255$. *Cette somme est assez fastidieuse à faire – une manière plus rapide consiste à observer que, en binaire : 11111111 + 1 = 100000000 (pour vous en convaincre, allez voir la section suivante !), et que* $100000000_2 = (2^8)_{10} = 256_{10}$

Pour 32 bits, utilisons directement la manière rapide ci-dessus : $2^{32} = 4294967296$ *; donc le nombre maximal que l'on peut coder sur 4 octets est 4294967291.*

Exercice 13: et à l'inverse...

Combien de bits sont nécessaires pour coder l'entier 1 ? L'entier 7 ? L'entier 200 ?

— *L'entier 1 se code sur 1 bit ;*
— *L'entier 7 sur 3 bits (puisque 3 bits, comme on l'a vu à la question précé-*

dente, peuvent justement coder jusqu'à $2^3 - 1 = 8 - 1 = 7$);
— *L'entier 200 sur 8 bits (qui ont, on l'a vu, un maximum de 255);*
— *Plus généralement, l'entier N peut se coder sur k bits, où k est tel que : $2^{k-1} \leq N < 2^k$.*

Bienvenue au CP ! Commençons par les additions

Il est essentiel pour la suite de bien comprendre comment fonctionnent les additions en binaire :

— 0 + 0 = 0 (sans retenue)
— 1 + 0 = 1 (sans retenue)
— 1 + 1 = 0 avec une retenue de 1
— (et attention :) s'il y avait une retenue préalable, 1 + 1 = 1 avec une retenue de 1

Ainsi considérons l'addition des deux nombres binaires 01011011 et 00100101 :

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

Exercice 14: additionnons donc !

Effectuez l'addition binaire des nombres 10101010 et 00010111. Ensuite, additionnez 10101010 et 11101010. Combien de bits sont nécessaires pour stocker les résultats de ces additions ?

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

Tous ces nombres peuvent être codés sur 8 bits (donc 1 octet) à l'exception du dernier où la somme de deux nombres codés sur 1 octet dépasse la valeur maximale possible (255 en décimal) et son codage <u>dépasse donc la capacité</u> d'un octet.

Dépassement de capacité en Python

Exercice 15: bonne nouvelle ! On passe (brièvement...) sur ordinateur !

Créez un script IDLE dans lequel vous mettez le code suivant et exécutez-le :

```python
# on importe d'abord une bibliotheque celebre en python,
# tres utilisee pour la rapidite des calculs
import numpy

# on utilise sa fonction uint8 pour coder l'entier 250 sur 8 bits
a = numpy.uint8 (250)
print (bin (a))

for i in range (1, 10):
    print (f" Iteration {i},valeur de a : {a}")
    print (f"en binaire : {bin (a)}")
    a += numpy.uint8 (1)
```

Que remarquez-vous après avoir exécuté ce code ? Pouvez-vous expliquer pourquoi cela se produit en utilisant les concepts abordés dans ce cours ?

La sortie de ce programme commence normalement :

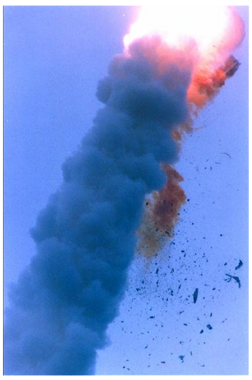
```
Iteration 1, valeur de a : 250
en binaire : 0b11111010
```

Puis cela se poursuit jusqu'à obtenir une erreur ressemblant à ceci :

```
RuntimeWarning: overflow encountered in ubyte_scalars
a += numpy.uint8 (1)
```

Ce qui se passe ici est un dépassement de capacité : on a demandé à l'ordinateur de stocker la valeur d'une variable (`a`) sur un espace mémoire limité (8 bits, donc un octet), puis on a fait augmenter la valeur de `a` jusqu'à dépasser le nombre maximal que l'on peut coder sur 8 bits et donc générer cette erreur.

Pourquoi est-ce important, ce genre de considération ? Demandez donc ça aux scientifiques qui travaillaient sur le premier vol de la fusée Ariane 5 à Kourou, en Guyane, le 4 juin 1996, et ont vécu ces 36 secondes tragiques... (Pour en savoir plus sur "la ligne de code la plus chère de l'Histoire" : page Wikipedia)



4 Encodage des entiers relatifs

4.1 Utilisation d'un bit de signe

On sait donc maintenant coder en binaire des entiers positifs – mais comment faire pour des entiers négatifs (appartenant à l'ensemble des entiers relatifs, noté $\mathbb{Z}$) ?

Une première idée consisterait à utiliser ce qu'on appelle un "bit de signe" : décider que lorsqu'on utilise une représentation sur 8 bits pour coder un entier signé, le bit le plus à gauche est dédié au signe du nombre : un "0" indique un nombre positif et un "1" un nombre négatif. De ce fait, seuls les 7 bits restants sont utilisés pour représenter la valeur absolue de l'entier. Ainsi, dans ce système, la plus grande valeur positive que l'on peut représenter n'est pas "11111111" (qui serait interprété comme un nombre négatif à cause du bit de signe à "1"). Au lieu de cela, la plus grande valeur positive possible est "01111111", ce qui correspond à 127 en notation décimale.

*Vocabulaire : on appelle le bit le plus à gauche d'une séquence le "**bit de poids fort**" et celui le plus à droite le "**bit de poids faible**" – correspondant aux puissances de deux plus élevées à gauche qu'à droite.*

Donc, par exemple, 4 sera codé 00000100 et -4 10000100, et si on fait la somme des deux :

$$\begin{array}{c} 0\,0\,0\,0\,0\,1\,0\,0 \\ 1\,0\,0\,0\,0\,1\,0\,0 \\ \hline 1\,0\,0\,0\,1\,0\,0\,0 \end{array}$$

Ce qui, exprimé en décimal, donnerait : 4 + (-4) = -8 – ce qui est un peu un problème... Face à cette limitation, d'autres méthodes de représentation des nombres négatifs ont été développées pour permettre desopérations arithmétiques plus simples et plus précises.

4.2 Le complément à 1 – puis à 2

Complément à 1

Le complément à 1 d'un nombre binaire est le nombre qui change tous les 0 en 1 et tous les 1 en 0. Si n est un nombre binaire, le complément à 1 de n est noté $\overline{n}$. Par exemple, le complément à 1 de 0011 est 1100.

Donc, par exemple, si fait la somme de 4 (00000100) et de $\overline{4}$, toujours sur 8 bits, on obtient :

$$\begin{array}{c} 0\,0\,0\,0\,0\,1\,0\,0 \\ 1\,1\,1\,1\,1\,0\,1\,1 \\ \hline 1\,1\,1\,1\,1\,1\,1\,1 \end{array}$$

Ce n'est évidemment pas ce qu'on cherche – mais on sent que c'est un début de piste...

Exercice 15: faisons une petite pause Python...

Ecrivez, dans IDLE, une fonction CompUn(Lst) qui prend en entrée une liste de bits (par exemple `[1,0,1,1,0,1]`) et retourne en sortie une liste contenant le complément à 1 (donc "inverse" tous les bits) de la liste en entrée (donc dans le cas précédent : `[0,1,0,0,1,0]`). Testez votre fonction sur les exemples suivants (la flèche représentant évidemment ce que doit retourner votre fonction) :

a. `CompUn([0])` $\rightarrow$ `[1]`

b. `CompUn([1])` $\rightarrow$ `[0]`

c. `CompUn([0, 1])` $\rightarrow$ `[1, 0]`

d. `CompUn([0,0,1,1,1])` $\rightarrow$ `[1,1,0,0,0]`

```python
def CompUn(Lst):
    '''Fonction qui va renvoyer le complement a 1 du nombre binaire
    passe en entree sous forme de liste'''

    # On initialise le resultat: une liste vide
    resultat = []

    # On boucle sur la totalite de la liste passee en argument
    for i in range(len(Lst)):
        # Pour chaque element de la liste,on ajoute son contraire dans la
        # liste resultat
        if Lst[i] == 0:
            resultat.append(1)
        else:
            resultat.append(0)

    # Et pour finir on renvoie la liste resultat ainsi completee
    return resultat
```

Complément à 2

Ce que l'on va utiliser pour encoder les nombres relatifs est la méthode du **complément à 2** que l'on peut énoncer de la manière suivante :

- Le bit de poids fort est toujours utilisé pour représenter le signe ;
- La représentation des nombres positifs est inchangée ;
- Les nombres négatifs sont le complément à 2 de leur valeur positive :
  - On effectue le complément à 1 du nombre ;
  - On ajoute 1.

Exemples :

- Codage de -3 sur 4 bits :
  - $3_{10} = 011_2$ (on laisse le bit de poids fort de côté puisqu'il est utilisé pour le signe)
  - Complément à 1 : 100
  - Complément à 2 : $100 + 1 = 101$
  - Donc le codage de -3 est, avec le bit de signe, est le "mot binaire" : 1101
- Codage de -4 :

  - $4_{10} = 100_2$ (on laisse le bit de poids fort de côté puisqu'il est utilisé pour le signe)
  - Complément à 1 : 011
  - Complément à 2 : 011 + 1 = 100
  - Donc le codage de -4 est, avec le bit de signe : 1100
- Codage de -1 :
  - $1_{10} = 001_2$
  - Complément à 1 : 110
  - Complément à 2 : 110 + 1 = 111
  - Donc le codage de -1 est, avec le bit de signe : 1111

Quel est l'intérêt ? Essayons de faire quelques sommes de chiffres opposés ***en ignorant la retenue finale*** :

$$\begin{array}{l} 0\,0\,1\,1 \\ 1\,1\,0\,1 \\ \hline 0\,0\,0\,0 \end{array}$$

$$\begin{array}{l} 0\,1\,0\,0 \\ 1\,1\,0\,0 \\ \hline 0\,0\,0\,0 \end{array}$$

Exercice 16: additions d'entiers relatifs

En utilisant le complément à deux et toujours sur 4 bits, effectuez les additions binaires suivantes

a. $-1_{10} + 1_{10}$
b. $2_{10} + -2_{10}$
c. $1_{10} + -4_{10}$
d. $-2_{10} + -4_{10}$
e. $2_{10} + 3_{10}$

a. $-1_{10} + 1_{10}$ *: On vient de voir l'encodage de -1 en complément à 2 donc on peut directement poser l'opération (toujours sur 4 bits) :*

$$\begin{array}{l} 1\,1\,1\,1 \\ 0\,0\,0\,1 \\ \hline 0\,0\,0\,0 \end{array}$$

b. $-2_{10} + -2_{10}$ *: Codage de -2 :*

- $2_{10} = 010_2$
- *Complément à 1 : 101*
- *Complément à 2 : 101 + 1 = 110*
- *Donc le codage de -2 est, avec le bit de signe : 1110*

$$\begin{array}{l} 1\,1\,1\,0 \\ 0\,0\,1\,0 \\ \hline 0\,0\,0\,0 \end{array}$$

c. $1_{10} + -4_{10}$ *: On a déjà l'encodage de -4 en complément à 2 donc on peut* *directement poser l'opération :*

$$\begin{array}{l} 0\,0\,0\,1 \\ 1\,1\,0\,0 \\ \hline 1\,1\,0\,1 \end{array}$$

Ce qui, on le verra dans le tableau plus bas, est bien le mot binaire correspondant à la valeur décimale -3*.*

d. $-2_{10} + -4_{10}$ *:*

$$\begin{array}{l} 1\,1\,1\,0 \\ 1\,1\,0\,0 \\ \hline 1\,0\,1\,0 \end{array}$$

Ce qui, toujours en vérifiant dans le tableau plus bas, correspond bien à la valeur décimale -6*.*

e. $2_{10} + 3_{10}$ *:*

$$\begin{array}{l} 0\,0\,1\,0 \\ 0\,0\,1\,1 \\ \hline 0\,1\,0\,1 \end{array}$$

(ce qui correspond bien à $+5$ *– juste pour confirmer que les additions d'entiers positifs fonctionnent toujours bien)*

Important:

On notera bien que dans la totalité des exemples précédents où on terminait l'addition avec une retenue (donc dans les cas où la somme "réelle" des deux binaires donnait un résultat $> 1111_2$*) on ignorait cette retenue - en d'autres termes on ne considérait que les 4 bits les plus à droite (donc de poids le plus faible) pour obtenir le résultat. Ces exemples étaient le a), le b), et le d). On peut aisément vérifier qu'il s'agit des exemples où soit on additionne deux valeurs négatives, soit on additionne une valeur positive à une négative de valeur absolue égale ou inférieure.*

On constate qu'avec cet encodage, pour tous mots binaires m_1 et m_2, on peut les additionner en effectuant simplement une addition binaire sans se soucier de leur signe et on obtiendra bien le résultat voulu si on ignore une éventuelle retenue finale.

En appliquant la méthode du complément à 2 encodons sur 4 bits le plus possible d'entiers relatifs – on obtient :

Mot Binaire				Entier Relatif
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	-8
1	0	0	1	-7
1	0	1	0	-6
1	0	1	1	-5
1	1	0	0	-4
1	1	0	1	-3
1	1	1	0	-2
1	1	1	1	-1

On constate qu'on a représenté ici, sur 4 bits, les entiers allant de -8 à $+7$; soit les entiers allant de $-(2^3)$ à (2^3-1). On peut facilement se convaincre que cette formule est généralisable en disant que sur k bits, on peut représenter des valeurs d'entiers relatifs allant de $-(2^{k-1})$ à $(2^{k-1}-1)$ — le complément à 2 de l'entier 0 servant à représenter l'entier $-(2^{k-1})$.

Réciproque : passer d'une représentation binaire à l'entier relatif

Pour convertir un nombre binaire stocké sur un octet en complément à deux vers sa représentation en entier relatif, suivez les étapes suivantes :

1. Si le bit le plus à gauche (bit de signe) est 0, le nombre est positif – il suffit de convertir les 7 bits restants directement de base 2 à base 10.
2. Sinon, le nombre est négatif :
   (a) Inversez tous les bits.
   (b) Ajoutez 1 au résultat.
   (c) Convertissez les 7 bits restants en base 10 et ajoutez un signe négatif devant le résultat.

Exercice 17: conversions en décimal

Convertissez en décimal les nombres représentés en utilisant la méthode du complément à deux suivant

a. 00001000

b. 11111101

c. 11111000

a. 00001000 : le bit de poids fort (le plus à gauche) est 0, donc c'est un entier positif, donc on convertit directement de binaire à décimal : en l'occurence on voit que ce nombre est égal à 2^3 soit 8.

b. 11111101 : cette fois, le nombre est négatif :

- *Inversion des bits / complément à 1 : 00000010*
- *Ajout de 1 : 00000011*

— *Conversion en base 10 :* $11_2 = 3_{10}$
— *Ajout du signe moins – le résultat est :* -3.

Note : pour vous en convaincre, vous pouvez poser l'addition 11111101 + 000000011 et vérifier que vous retombez bien sur 0 (en ignorant bien sûr la retenue finale).

c. *11111000 : même démarche :*

— *Inversion des bits / complément à 1 : 00000111*
— *Ajout de 1 : 00001000*
— *Conversion en base 10 :* $1000_2 = 8_{10}$
— *Ajout du signe moins – le résultat est :* -8.

5 Représentation approximative des nombres réels

ATTENTION:

Ce chapitre est assez complexe, notamment au niveau calculatoire. Il est important que vous en compreniez les concepts et les modes de calcul, mais **il ne vous sera pas demandé de connaître précisément la norme IEEE 754 – c'est hors programme**. C'est une des raisons pour lesquelles ce chapitre contient relativement moins d'exercices que les autres.

Exercice 18: commençons par essayer de voir le problème

Dans une console Python, effectuez les calculs suivants :

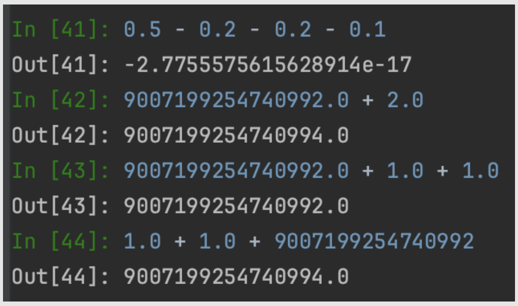
a. `0.5 - 0.2 - 0.2 - 0.1`

b. `9007199254740992.0 + 2.0`

c. `9007199254740992.0 + 1.0 + 1.0`

d. `1.0 + 1.0 + 9007199254740992.0`

L'exécution des commandes ci-dessus en console Python donnent le résultat suivant :



```
In [41]: 0.5 - 0.2 - 0.2 - 0.1
Out[41]: -2.7755575615628914e-17
In [42]: 9007199254740992.0 + 2.0
Out[42]: 9007199254740994.0
In [43]: 9007199254740992.0 + 1.0 + 1.0
Out[43]: 9007199254740992.0
In [44]: 1.0 + 1.0 + 9007199254740992
Out[44]: 9007199254740994.0
```

Je pense que les erreurs et incohérences sautent aux yeux...

La raison de ces erreurs est qu'il n'est pas possible de représenter la totalité des nombres réels (de l'ensemble mathématique $\mathbb{R}$) avec exactitude en binaire – ce qui en première analyse se comprend intuitivement très bien en constatant que le nombre de réels compris entre 0 et 1 est infini, tandis que le nombre de combinaisons de bits sur une taille mémoire donnée est nécessairement finie. On doit donc recourir à des approximations – et l'objet de ce chapitre est de comprendre comment ces approximations fonctionnent.

5.1 Puissances négatives de 2

Jusqu'à présent, nous avons travaillé avec des puissances successives de deux toujours positives – 1, 2, 4, 8, 16, 32, etc... – et en avons considéré la somme pour convertir des nombres de la base 2 à la base 10 – par exemple :

$$\begin{aligned} 10_{10} &= (8+2)_{10} \\ &= (1\times 2^3 + 0\times 2^2 + 1\times 2^1 + 0\times 2^0)_{10} \\ &= (1010)_2 \end{aligned}$$

Nous allons ici étendre cette notion aux puissances *négatives* de 2 pour représenter des nombres compris dans l'intervalle [0, 1[:

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	...
0.5	0.25	0.125	0.0625	0.03125	...

Ainsi, en utilisant cette notation, le mot binaire 1010 codé sur 4 bits aura pour valeur :

2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	0	1	0

Que l'on va calculer ainsi :

$$\begin{aligned} 1010_2 &= (1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4})_{10} \\ &= (0,5 + 0,125)_{10} \\ &= (0,625)_2 \end{aligned}$$

5.2 La notation scientifique

On utilise beaucoup, en mathématiques et en sciences, ce qu'on appelle la *notation scientifique* des nombres décimaux. Celle-ci se compose d'un signe (+ ou -) ; d'une *mantisse*, notée m et comprise dans l'intervalle [1, 10[; et d'un *exposant*, nombre entier relatif noté n. L'écriture de cette valeur est alors $\pm m \times 10^n$. Ainsi, spécifiquement :

$$\begin{array}{rll} 2\,156 & \text{s'écrit} & +2,156 \times 10^3 \\ -398\,879,62 & \text{s'écrit} & -3,9887962 \times 10^5 \\ 0,000142 & \text{s'écrit} & +1,42 \times 10^{-4} \\ 1,34 & \text{s'écrit} & +1,34 \times 10^0 \end{array}$$

5.3 La norme IEEE 754

Cette norme, développée dans les années 1980 aux USA, existe en deux versions, l'une, dite "codage simple précision", pour une représentation sur 32 bits (4 octets), et l'autre, dite "codage double précision", pour une représentation sur 64 bits (8 octets). Dans les deux cas elle s'appuie sur le même principe que la notation scientifique – à s'avoir une décomposition du nombre en une somme de puissances, utilise les puissances négatives de 2 pour décrire le plus précisément possible la mantisse, et s'écrit sous la forme suivante :

$$(-1)^s \cdot (1 + m) \times 2^{(n-d)}$$

— s est le bit de signe, valant 0 ou 1 ;

— m est la mantisse, comprise (puisqu'on raisonne ici non plus en base 10 mais en base 2) dans l'intervalle [0, 1[;

— n est toujours l'exposant, mais il est cette fois "*décalé*" (on peut également dire "*biaisé*") d'une valeur d qui dépend du format choisi (32 ou 64 bits).

Ecrite de la sorte en mémoire et encodée sur 32 bits, la valeur d'un nombre flottant sera structurée comme suit :

$$\overbrace{\underbrace{\text{Signe}}_{s}}^{(1\ bit)} \quad \overbrace{\underbrace{\text{Exposant}}_{n}}^{(8\ bits)} \quad \overbrace{\underbrace{\text{Mantisse}}_{m}}^{(23\ bits)}$$

Voyons un exemple concret, le mot M de 32 bits suivant :

11000011010101101100000000000000

Commençons par le décomposer en signe (bit de poids fort), exposant (8 bits suivants), et mantisse (23 bits restants) :

$$\overbrace{1}^{signe} \quad \overbrace{10000110}^{exposant} \quad \overbrace{10101101100000000000000}^{mantisse}$$

Calculons alors sa valeur décimale :

$$\begin{aligned}
\text{signe} &= -1^1 \\
&= -1 \\
\\
\text{exposant} &= 2^7 + 2^2 + 2^1 \\
&= 128 + 4 + 2 \\
&= 134 \\
\\
\text{mantisse} &= 2^{-1} + 2^{-3} + 2^{-5} + 2^{-6} + 2^{-8} + 2^{-9} \\
&= 0,677734375
\end{aligned}$$

Appliquons à présent la formule générale décrite plus haut $((-1)^s \cdot (1+m) \times 2^{(n-d)})$ en utilisant un décalage d de 127 puisque nous étudions un encodage sur 32 bits :

$$\begin{aligned}
M &= -1 \times (1 + 0,677734375) \times 2^{(134-127)} \\
&= -1,677734375 \times 2^7 \\
&= -214,75
\end{aligned}$$

Exercice 19: Entrainons-nous...

Convertissons quelques réels codés sur 32 bits pour s'entrainer...

a.

0	1	0	0	0	0	0	0	1	1	0	1	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

b.

1	0	1	1	1	1	1	1	0	1	1	1	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

a.

0	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*1*	*1*	*0*	*1*	*0*	*1*	*0*	*0*
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*

Isolons tout d'abord les différentes composantes de ce mot binaire :

- *Signe : 0*
- *Exposant : 10000001*
- *Mantisse : 10101000000000000000000*

Comme ci-dessus, calculons à présent les valeurs de ces différentes com-

posantes :

$$\begin{aligned} signe &= -1^0 \\ &= +1 \\ \\ exposant &= 2^7 + 2^0 \\ &= 128 + 1 \\ &= 129 \\ \\ mantisse &= 2^{-1} + 2^{-3} + 2^{-5} \\ &= 0,65625 \end{aligned}$$

Appliquons enfin la formule générale décrite plus haut ($(-1)^s \cdot (1+m) \times 2^{(n-d)}$) *en utilisant un décalage d de 127 puisque nous étudions un encodage sur 32 bits :*

$$\begin{aligned} M &= 1 \times (1 + 0,65625) \times 2^{(129-127)} \\ &= 1,65625 \times 2^2 \\ &= 6,625 \end{aligned}$$

b.

1	*0*	*1*	*1*	*1*	*1*	*1*	*1*	*0*	*1*	*1*	*1*	*1*	*0*	*0*	*0*
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*

Isolons tout d'abord les différentes composantes de ce mot binaire :

- *Signe : 1*
- *Exposant : 01111110*
- *Mantisse : 11110000000000000000000*

Comme ci-dessus, calculons à présent les valeurs de ces différentes composantes :

$$\begin{aligned} signe &= -1^1 \\ &= -1 \\ \\ exposant &= 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 \\ &= 64 + 32 + 16 + 8 + 4 + 2 \\ &= 125 \\ \\ mantisse &= 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} \\ &= 0,9375 \end{aligned}$$

Appliquons enfin la formule générale décrite plus haut ($(-1)^s \cdot (1+m) \times 2^{(n-d)}$) *en utilisant un décalage d de 127 puisque nous étudions un encodage sur 32 bits :*

$$\begin{aligned} M &= -1 \times (1 + 0,9375) \times 2^{(125-127)} \\ &= -1,9375 \times 2^{-2} \\ &= -1,9375 \times 0,25 \\ &= 0,484375 \end{aligned}$$

Exercice 20: C'est un peu fastidieux... et si on codait ça ?

a. Sur papier, écrire en pseudo-code l'algorithme d'une fonction qui prend en entrée une liste de 32 éléments représentant les 32 bits encodant un nombre flottant et qui calcule puis renvoie la valeur décimal du réel correspondant.

b. Dans IDLE, écrire cette fonction en Python. Si vous avez terminé, rendez cette fonction robuste – qu'elle sache gérer des listes n'ayant pas le bon nombre d'éléments, ou ne contenant pas que des 0 ou des 1...

A REDIGER !

5.4 Alors d'où viennent les erreurs qu'on a vues ?

Le problème est qu'il n'est en général pas possible de coder *exactement* un nombre réel avec ce système – sauf si, par chance, il peut se décomposer exactement de cette manière. L'ordinateur en est donc réduit à stocker des approximations des valeurs qu'on lui donne.

Par exemple : le nombre flottant le plus proche de 1,2 est

$$0 - 01111111 - 00110011001100110011001$$

La valeur exacte de ce nombre, si on le développe comme on l'a fait plus tôt, est de 1.2000000476837158 – ce qui est évidemment une différence négligeable pour la représentation de valeurs à l'échelle humaine (qu'il s'agisse, de distances, de pourcentages dans des contextes divers, de montants d'argent, de mesures physiques telles que le poids ou la vitesse d'un objet...). Cependant, il faut bien avoir conscience de l'existence de ces approximations.

Deux remarques :

— Il est évident ici que l'ordinateur utilise une méthode d'arrondis pour passer d'un mot binaire à sa valeur décimale et réciproquement. Cette méthode est hors programme.

— Si cependant c'est un sujet qui vous intéresse n'hésitez pas à consulter ce site pour voir plus en détails comment différentes valeurs réelles sont effectivement codées au moyen de la norme IEEE 754.

En tout état de cause, l'objet de ce chapitre est principalement de vous alerter sur deux points fondamentaux :

— Le fait **qu'il ne faut jamais** effectuer des tests d'égalité entre nombres flottants dans un programme (on l'a vu en début de chapitre, et on en avait vu également des exemples dans le cours sur Python) ;

— Le fait que dans des contextes où les besoins de précision sont hors du commun (dans certaines disciplines scientifiques par exemple), il sera nécessaire de trouver des moyens de contourner ces difficultés pour éviter de générer de réelles erreurs.

6 Représentation des caractères

ATTENTION:
Comme le chapitre précédent, celui-ci a pour vocation d'être une introduction au codage des caractères – vous devez en comprendre les principes et les mécanismes, mais **il ne vous sera pas demandé de connaître précisément le normes d'encodage (ASCII et Unicode) – c'est hors programme**.

6.1 Code ASCII

Les ordinateurs ne stockent que des 0 et des 1 ; on sait assez simplement établir une correspondance entre un nombre décimal et sa traduction en binaire. Partant de ces deux constats, on a initialement établi une correspondance unique entre chaque caractère et un entier naturel – on appelle cette démarche le *charset*.

Ceci a donné naissance en 1963 au code ASCII (*American Standard Code for Information Interchange*), toujours en usage, et permettant d'encoder 128 caractères : 33 caractères dits de contrôle (codes 0 à 32 et code 127 – ils incluent des caractères "blancs" comme les espaces ou les tabulations, des suppressions, des retours à la ligne, des marqueurs de début et de fin de texte...), 94 caractères standards (codes 33 à 126) comportant les 26 lettres minuscules, les 26 lettres majuscules, les 10 chiffres, et 32 symboles (ponctuation, symboles arithmétiques...).

Le tableau ci-dessous vous présente tout ceci exhaustivement (pour votre culture générale – il est évident **que ce n'est pas à connaitre par cœur !**).

Code	Caractères	Code	Caractères	Code	Caractères	Code	Caractères
0	NUL	32	Espace	64	@	96	‘
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	’	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	ˆ	126	˜
31	US	63	?	95	_	127	DEL

→*A première vue, qu'est-ce qui ne va pas ce tableau – quel(s) problème(s) va-t-on rapidement rencontrer ??*

Il manque énormément de caractères – et d'autant plus si on n'est pas anglophone ! Déjà, pour les francophones, on note qu'il n'y a aucun accent. Mais que devaient penser les Arabes, les Chinois, les Russes... ?

→*Sur combien de bits va-t-on encoder ces caractères ASCII ??*

On compte 128 caractères soit (et ce n'est évidemment pas un hasard) 2^7. Il serait donc possible de n'utiliser que 7 bits pour coder les caractères ASCII.

En pratique, on a toujours codé ces caractères sur un octet (donc 8 bits) en utilisant à l'origine le bit de poids fort comme un mécanisme de contrôle d'erreur (qui étaient fréquentes avec les premières mémoires électroniques). Le principe était d'en faire un bit de parité : il était positionné à 0 ou à 1 de manière à ce *que tous les caractères aient toujours un nombre pair de bits à 1 ; et si ce n'était pas le cas, une erreur était détectée.*