Chapitre 5

Représentation des données

En informatique, toutes les données et les programmes sont stockés à l'aide de 0 et de 1, appelés **bit** pour Binary Digit. Physiquement, un 1 est représenté par une tension électrique de 5V et le 0 par 0V.

On regroupe souvent ces bits par paquets de 8 que l'on appelle **octet**. Il y a donc $2^8 = 256$ octets distincts.

Les ordinateurs actuels sont quasiment tous qualifiés de 64-bits, c'est-à-dire qu'ils peuvent manipuler simultanément 64 bits, soit 8 octets à la fois.

Si dans une case mémoire, on trouve l'octet `01011001`, à quoi correspond-il ? Et bien, on ne sait absolument pas, sa signification dépend du contexte.

5.1 Encodage de caractères

La première norme pour les chaînes de caractères a été l'ASCII (American Standard Code for Information Interchange). Elle est parue dans les années 1960. Elle est constituée d'un tableau de 127 caractères (codés sur 7 bits). Sur ces 127 caractères, seuls 95 sont imprimables, les autres sont des caractères de contrôle comme par exemple le retour à la ligne `\n`, la tabulation `\t`, le retour arrière `\b` ou la cloche `\a`.

Tant qu'on écrit en anglais, ces caractères sont suffisants mais il a fallu ajouter les caractères accentués, les lettres grecques, cyrilliques, arabes, asiatiques,...

D'autres normes ont donc été développées, d'abord localement avec les normes ISO-8859. Une norme universelle a aussi vu le jour : UTF-8. C'est actuellement la norme la plus utilisées dans le monde [1].

Quand une chaîne de caractère est lue, il faut que l'ordinateur sache la norme utilisée sinon il interprète mal le texte. C'est le rôle du paramètre `encoding` quand on ouvre un fichier texte.

5.2 Nombres entiers

La représentation des nombres se base sur le binaire et la base 2.

5.2.1 Entiers positifs

Un nombre entier positif peut s'écrire comme une somme de puissances de 2. Cette somme est unique, à l'ordre près des termes. Par exemple, $39 = 32 + 4 + 2 + 1 = 2^5 + 2^2 + 2^1 + 2^0$. On note l'écriture binaire $39 = (100111)_2$.

Dans un ordinateur, on représente donc les entiers en écriture binaire. Il y a deux conventions [2] :

1. d'après wikipedia, en octobre 2020, 95,2% des sites web utilisaient l'UTF-8
2. en fait, il y en a plus mais ce sont les principales

big-endian : le poids fort devant : $\overline{\texttt{100111}}$. C'est la notation qu'on utilise à la main.

little-endian : le poids faible devant : $\overline{\texttt{111001}}$. C'est la notation inverse. A l'époque des premiers ordinateurs, cette convention permettait de câbler plus facilement l'addition.

Dans la suite du cours, nous utiliserons la notation plus naturelle big-endian. En Python, on note `0b100111`. Les caractères `0b` permettent de préciser que le nombre est écrit en binaire. Il existe d'autres préfixes comme par exemple `0x` pour écrire en hexadécimal.

Les calculs se font en posant les opérations comme en primaire, mais en binaire. Il faut donc utiliser les tables d'addition et de multiplication suivantes :

+	0	1
0		
1		

×	0	1
0		
1		

Exercice 16.
Effectuer les calculs suivants :

	0	0	1	1	0	1	0	1
+	0	1	1	1	0	1	0	0
	.	.	.	.	.	.	.	.

	0	0	0	0	1	1	0	1
×	0	0	0	0	1	0	1	1
	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.

Convertir ces nombres en base 10 et vérifier les calculs :
Premier calcul : Deuxième calcul :

5.2.2 Entiers signés

Pour les entiers signés (avec un éventuel signe négatif), il a fallu réfléchir pour conserver le circuit électrique de l'addition des nombres entiers positifs.

La seule solution est la convention dite "complément à 2 plus 1". Pour l'exemple, travaillons sur un octet et écrivons la représentation de -89.
D'abord, $89 = 64 + 16 + 8 + 1 = 2^6 + 2^4 + 2^3 + 1 =$ `0b01011001`
Ensuite, on passe au complémentaire à 2 en inversant tous les bits : `0b10100110`.
Enfin, on ajoute 1 : `0b10100111`.
On obtient alors le codage de -89 : $-89 =$ `0b10100111`.

Pour le chemin inverse, on a besoin de soustraire 1, c'est-à-dire d'additionner par (-1). Par le même principe, on trouve que

$1 =$ `0b________` $\underset{compl}{\rightsquigarrow}$ `0b________` $\underset{+1}{\rightsquigarrow}$ $-1 =$ `0b________`

Ainsi `0b11100011` $\underset{-1}{\rightsquigarrow}$ `0b________` $\underset{compl}{\rightsquigarrow}$ `0b________` =

Ce qui signifie que `0b11100011` représente le nombre :

Vérifions que l'addition est respectée, par exemple avec (-89)+57 : $57 =$ `0b________`

	1	0	1	0	0	1	1	1
+	.	.	.	.	.	.	.	.
	.	.	.	.	.	.	.	.

et

`0b________` $\underset{-1}{\rightsquigarrow}$ `0b________` $\underset{compl}{\rightsquigarrow}$ `0b________` =

donc on a bien (-89)+57 =

Remarques

- On a un test assez pratique et rapide pour tester si un nombre entier signé est négatif : la valeur du premier bit est à 1.
- Les nombres accessibles avec cette convention vont de -128 à 127 sur un octet. Pour des nombres sur 64 bits, on a donc accès aux nombres entre -2^{63} à $2^{63}-1$.
- Pour un système sur un octet, puisque `0b01111111 + 0b00000001 == 0b10000000`, on en déduit que 127 + 1 = -128. On appelle ce problème un dépassement des capacités de calcul. Ça a été à l'origine de nombreux bugs !
  Nous n'aurons jamais ce soucis avec Python car il ne fonctionne pas sur un nombre fixe de bits...

5.3 Nombres flottants

Pour les nombres à virgule, il existe une représentation de la même façon que pour la base 10 :

$$13,375 = 1\cdot 10^1 + 3\cdot 10^0 + 3\cdot 10^{-1} + 7\cdot 10^{-2} + 5\cdot 10^{-3}$$
$$13,375 = 8+4+1+0,25+0,125 = 2^3+2^2+2^0+2^{-2}+2^{-3} = (1101,011)_2$$

Cette représentation n'est pas très efficace car l'amplitude des nombres représentés n'est alors pas très importante et il faut trouver une façon de stocker la position de la virgule.

On préfère la représentation en virgule flottante, basée sur la notation scientifique. Python permet d'écrire directement des nombres à virgule basé sur cette notation : `1.3375e2 == 133,75` ou `22.57e-3 == 0.02257`.

Par contre, pour représenter ces nombres en mémoire, on utilise la puissance 2 puisqu'elle est à la base de toute l'informatique. Ainsi

$$13,375 = 1,3375\cdot 10^1 \qquad \text{(écriture scientifique)}$$
$$(1101,011)_2 = 1,101011\cdot 2^3 \qquad \text{(écriture scientifique en base 2)}$$

Le 3 est l'**exposant** et le 1,101011 est appelé la **mantisse**.

La convention sur 64 bits est alors la suivante :

- 1 bit pour le signe : 0 pour un nombre positif, 1 pour un nombre négatif.
- 11 bits pour l'exposant : avec 11 bits on a 2048 possibilités. On stocke en réalité (exposant + 1023). On a ainsi des valeurs possibles de l'exposant entre -1023 et 1024
- 52 bits pour la mantisse : Puisque cette mantisse commence nécessairement par 1, on ne stocke pas ce premier 1. Ça économise encore 1 bit.

Le nombre 13,375 est donc stocké :

signe	exposant	mantisse
0	10000000010	10101100...0

Exercice 17.

Dans chacun des cas, taper les tests dans python et commenter

1. `0.1 * 2 == 0.2` et `0.1 * 3 == 0.3`
2. `2.0 ** 53 == 2.0 ** 53 + 1`
3. `2.0 ** 560 == 2.0 ** 560 + 2.0 ** 500`
4. `0.3 + 0.2 + 0.1` et `0.2 + 0.1 + 0.3`

Conclusion de cette partie :

Il ne faut jamais tester l'égalité de deux flottants !

À la place, on test si les deux flottants sont très proches.

De même, si on a besoin de faire une grande addition avec des flottants, il est préférable de les additionner du plus petit (en valeur absolue) au plus grand. De cette façon, on minimise les approximations dues à la finitude de la mantisse.

5.4 opérations binaires

Python permet de manipuler directement les bits d'un entier et de faire des opérations logiques sur les entiers comme si c'était une suite de bits indépendants.

- Opérateur logique `AND` (`&` en Python) : chaque bit du résultat vaut 1 si et seulement si les 2 bits des opérandes sont à 1.
  Exemple : `0b1100 & 0b1010 == 0b1000`.
- Opérateur logique `OR` (`|` en Python) : chaque bit du résultat vaut 0 si et seulement si les 2 bits des opérandes sont à 0.
  Exemple : `0b1100 | 0b1010 == 0b1110`.
- Opérateur logique `XOR` (`^` en Python) : chaque bit du résultat vaut 0 si et seulement si les 2 bits des opérandes sont identiques.
  Exemple : `0b1100 ^ 0b1010 == 0b0110`.
- Opérateur de décalage vers la gauche (`<<` en Python) : chaque bit de l'opérande de gauche est décalé vers la gauche du nombre de bits de l'opérande de droite. Les bits qui dépassent la capacité de stockage sont purement et simplement perdus.
  Exemple : (si on se restreint à 8 bits) `0b00011011 << 4 == 0b10110000`.
  Sans perte de bit, un décalage de 1 vers la gauche correspond à une multiplication par 2.
- Opérateur de décalage vers la droite (`>>` en Python) : chaque bit de l'opérande de gauche est décalé vers la droite du nombre de bits de l'opérande de droite. Les bits qui dépassent la capacité de stockage sont purement et simplement perdus.
  Exemple : `0b11010 >> 2 == 0b00110`.
  Sans perte de bit, un décalage de 1 vers la droite correspond à une division par 2.

On peut ainsi stocker des informations binaires dans un seul et même grand nombre. Pour récupérer les bits, on manipule les opérations logiques en créant des masques.

Par exemple, pour savoir si le nombre 89 a son quatrième bit à 1, on crée le masque `1<<3` (un seul 1 en 4ème position) et on compare avec l'opérateur `AND` :

```python
nombre_a_tester = 89
masque = 1 << 3
if (nombre_a_tester & masque) == masque:
    print("4eme bit à 1")
else:
    print("4eme bit à 0")
## Affichera "4eme bit à 1" car 89 = 0b01011001
```

Exercice 18.

1. Créer les masques suivants : `0b10001001` et `0b1111111111`
2. Déterminer en utilisant un masque et une seule condition si 5687 a les bits 1 et 13 à 1.
3. Déterminer en utilisant un masque et une seule condition si 4810 a au moins un des bits 6 ou 7 à 1 (le bit 6, le bit 7 ou les deux).