Chapitre 4 : Représentation des nombres

I Nombres binaires

Au lieu de décomposer un entier en utilisant les puissances de 10 (unités, dizaines, centaines, etc...), on utilise les puissances de 2.

Théorème I.1 Tout nombre entier $n \in \mathbb{N}$ admet une décomposition unique en somme de puissances de 2 : il existe un $p \in \mathbb{N}$ et un unique $(b_0, b_1, \dots, b_{p-1}) \in \{0,1\}^p$ tels que :

$$n = b_0 2^0 + b_1 2^1 + \cdots + b_{p-1} 2^{p-1} = \sum_{k=0}^{p-1} b_k 2^k.$$

On notera alors $n = \overline{b_{p-1} \dots b_1 b_0}$.

Pour passer de l'écriture binaire à l'écriture en base 10, on ajoute les puissances de 2, par exemple : $\overline{110001}$ =

■ **Exemple 1**

- Les premiers entiers naturels s'écrivent en binaire :

- Les binaires de la forme $\overline{10\dots0}$ sont
- Les binaires de la forme $\overline{11\dots1}$ sont

■

En général, pour passer de l'écriture en base 10 à l'écriture binaire, on effectue les divisions euclidiennes successives des quotients par 2.

■ **Exemple 2** Décomposons 18 en binaire : ■

■ **Remarque 1**

- Pour additionner deux nombres en binaire, on pose l'addition comme pour la base 10. Par exemple,

- Multiplier un nombre par 2^k revient à
- Diviser un nombre par 2^k revient à

■

Sous python, on dispose de la fonction `bin` qui prend un entier et renvoie une chaîne de caractère donnant l'écriture binaire de l'entier. On peut aussi facilement reprogrammer cette fonction :

```
def binaire(n : int) -> str:
    '''


    '''
    s = ''



    return s
```

II Stockage de l'information

II.1 Unités de mémoire

Les données sont stockées et transitent sous forme de **bits** (**bi**nary digi**ts**). Un bit ne peut prendre que 2 valeurs : 0 ou 1.
Un **octet** est une cellule mémoire élémentaire composée de 8 bits :

- le premier bit s'appelle le **bit de poids fort** (**MSB**),
- le dernier bit s'appelle le **bit de poids faible** (**LSB**).

Pour mesurer la taille que prennent des données, il y a deux conventions d'unités :

Variante standard			**Variante binaire**		
Nom	**Unité**	**Valeur**	**Nom**	**Unité**	**Valeur**
kilooctet	ko	10^3o	kibioctet	kio	2^{10}o=1 024o
mégaoctet	Mo	10^6o	mébioctet	Mio	2^{20}o=1 048 576o
gigaoctet	Go	10^9o	gibioctet	Gio	2^{30}o
téraoctet	To	10^{12}o	tébioctet	Tio	2^{40}o
pétaoctet	Po	10^{15}o	pébioctet	Pio	2^{50}o

II.2 Représentation des entiers

II.2.1 Les entiers positifs sur des mots de taille fixe

Avec 1 octet, on peut écrire tous les nombres entiers de 0 à
Si on dispose de 4 octets, c'est-à-dire 32 bits on peut écrire tous le nombres de 0 à
L'avantage de fixer le nombre de bits utilisés pour stocker les nombres est de faciliter l'allocation de la mémoire : une valeur codée sur 4 octets occupe une plage mémoire bien définie qui ne débordera pas.
Le désavantage est qu'en cas de dépassement de la taille allouée aux entiers, on perd de l'information : par exemple, pour des entiers codés sur 32 bits, $2^{31} + 2^{31}$ donne 0 !

■ **Remarque 2** Ce problème n'est pas apparu lors de nos TP car Python gère nativement ces problèmes (voir un peu plus bas). Pour travailler avec des entiers non signés sur des mots de taille fixe, on peut utiliser les types `uint8`, `uint16`, `uint32` et `uint64` de la bibliothèque `numpy`. ■

II.2.2 Les entiers signés sur des mots de taille fixe

Su n bits, on représente un entier relatif $x \in [\![-2^{n-1}, 2^{n-1}-1]\!]$ en distinguant le signe :

- si $x \geq 0$, on stocke l'entier x ;
- si $x < 0$, on stocke l'entier $x + 2^n$.

■ **Remarque 3** Lorsque $x \geq 0$, le MSB vaut 0 et lorsque $x < 0$, il vaut 1.
On appelle cette représentation le **complément à deux**.
La méthode naïve (on stocke $|x|$ dans le deuxième cas) pose plusieurs soucis : on peut représenter 0 de deux façons et l'addition entre deux entiers devient plus compliquée à faire. ■

Pour un octet, on peut représenter tous les entiers de $-2^7 = -128$ à $2^7 - 1 = 127$. Par exemple,

- `10000000` représente
- `11111111` représente

Plus généralement, sur n bits,

- -2^{n-1} est représenté par :
- -1 est représenté par :
- -2^k (avec $k < n-1$) est représenté par :

En pratique, pour trouver la représentation de l'opposé d'un entier x :

- on change tous les bits : par exemple pour $x = 4 =$
- on rajoute 1 :

Pour calculer $3 - 4$, on fait :

En base 10	En machine	Sans complément
3_{10}		
-4_{10}		
$(-1)_{10}$		

Si on connaît la représentation en compléments à deux, par exemple `10101010`, on prend l'opposé :

- on change tous les bits :
- on ajoute 1 :

C'est donc :

■ **Remarque 4** Pour travailler avec des entiers signés sur des mots de taille fixe, on peut utiliser les types `int8`, `int16`, `int32` et `int64` de la bibliothèque `numpy`. On a toujours les mêmes soucis de dépassement. ■

II.2.3 Entiers multi-précision de Python

Le type `int` de Python permet d'éviter les problèmes de dépassement et la seule limite à la taille des entiers stockés est la mémoire disponible. Grossièrement, un entier est stocké sous forme d'un tableau contenant ses chiffres en représentation binaire.
Il ne faut toutefois pas perdre de vue que lorsque les entiers deviennent plus grands que ceux naturellement gérés par les processeurs, la complexité des opérations n'est plus vraiment constante...

III Représentation des réels

Pour représenter un nombre à virgule en machine, on passe de nouveau en base 2.

- Le nombre binaire $11,01_2$ vaut en décimal : $2^1 + 2^0 + \dfrac{1}{2^2} = 3,25$.
- Pour trouver la représentation binaire de $13,625$, on trouve celle de 13 et celle de $0,625$
  - ▷ $13 =$
  - ▷ Pour $0,625$, on fait des multiplications par 2 successives :

  Donc $13,625 =$

■ **Remarque 5**
- Un nombre décimal est un nombre réel dont l'écriture décimale est finie. Par exemple, $0,123$ est décimal, mais $\dfrac{1}{3}$ n'est pas décimal.
  Les nombres réels dont la décomposition binaire est finie sont des nombres décimaux, mais la réciproque n'est pas vraie ! Par exemple, pour $0,2$:
- Comme la taille en mémoire est finie, on ne pourra en général pas stocker exactement un réel, mais on n'en stockera qu'une valeur approchée.

■

Les nombres à virgule flottante se basent sur la représentation scientifique binaire des réels :

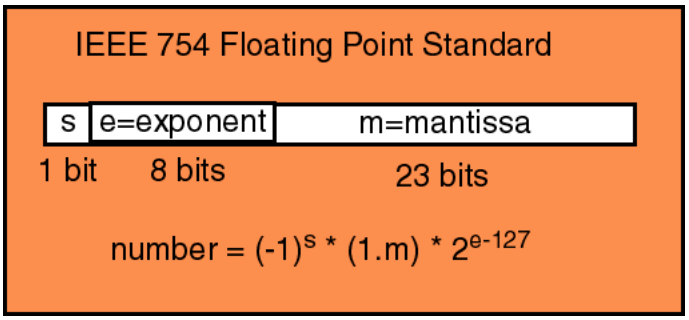
Théorème III.1 Soit x un réel non nul. Il existe un unique triplet $(s, m, e) \in \{-1, 1\} \times [1, 2[\times \mathbb{Z}$ tel que $x = s \times m \times 2^e$.
- s est le **signe** de x ;
- m est la **mantisse** ;
- e est **l'exposant**.

■ **Exemple 3** Par exemple, pour $x = -5,625$ ■

On peut remarquer que la mantisse est toujours sous la forme $\overline{1, b_1 b_2 \ldots}$.

Grâce à cette représentation, on peut stocker de la même façon des nombres très grands et très petits, sans être surchargé de caractères peu significatifs.

La norme IEEE 754 **simple précision** indique comment stocker un nombre à virgule flottante sur 32 bits :
- le signe sur le MSB ;
- l'exposant décalé sur les 8 bits suivants ;
- la mantisse-1 sur les 23 bits restants.



- l'exposant est décalé : on lui rajoute $2^7 - 1$ ($=\overline{01111111}$).
  Ainsi, un exposant `00000001` correspond en fait à $e = 1-(2^7-1) = -126$, qui est l'exposant le plus bas possible.
- L'exposant `00000000` (-127) est reservé pour 0 et les très petits nombres, et l'exposant `11111111` (128) est reservé pour les infinis et les NaN.
- Pour $x = -\overline{1,01101} \cdot 2^2$, on a
- Le flottant 1.0 vaut `0 01111111 00000000000000000000000` en machine.
  Le flottant suivant est `0 01111111 00000000000000000000001`, qui vaut $1 + 2^{-23}$, avec $2^{-23} \cong 1,2 \times 10^{-7}$.
- Le flottant suivant 2^{23} est $2^{23} + 1$!

En **double précision**, les flottants sont codés sur 64 bits avec 11 bits pour l'exposant (décalé de $2^{10} - 1$) et 52 bits pour la mantisse.

III.1 Attention, dangers !

1. Dépassement de capacité :
   - Pas de problème sous Python avec les entiers.
   - Pour les flottants représentés sur 64 bits, l'exposant maximal est 1023 et le minimal est -1022. Un calcul dont le résultat dépasse ces limites donne un dépassement arithmétique :
     - ▷ **Overflow** : le résultat est approximé à `inf`
     - ▷ **Underflow** : le résultat est approximé à `0`
2. Précision et arrondis : il est très rare qu'on puisse stocker un nombre réel exactement en machine.
   - La norme IEEE 754 impose que les nombres à virgules soient arrondis à la valeur représentable la plus proche : si le premier bit non représenté vaut 1 on arrondi vers le haut, sinon, on arrondi vers le bas.
   - Par exemple, le nombre $0,4 = \overline{0,011001100110011...}$ a un développement en base 2 infini. Il est approché par 0.4000000000000000222044604925031308084726333618164062 5. Sous Python, on obtient :

     ```
     >>>0.4 == 0.40000000000000002
     True
     ```

   - Lorsqu'on effectue des calculs avec des nombres dont les ordres de grandeurs sont très différents :

```
>>>1+2**(-54)-1
0.0
>>>1-1+2**(-54)
5.551115123125783e-17
```

On ne fait pas de test == ou != entre flottants !

IV Exercices

Exercice 1 1. Donner la valeur en base 10 des entiers relatifs représentés en complément à deux sur un octet par :
(a) `00001111` (b) `00010101` (c) `10010101` (d) `11111011` (e) `01000001`

2. Représenter en complément à 2 sur 16 bits les nombres 36000 et -42.
3. Soit $n \in \mathbb{N}$. Combien de chiffres a la représentation décimale de n ? Combien de chiffres a la représentation binaire de n ?
4. On pose `a = np.uint8(280)` et `b = np.uint8(240)` qui sont deux entiers de type `np.uint8` (entiers non signés codés sur 8 bits).
Que valent `a`, `b`, `a+b`, `a-b`, `a//b` et `a/b` ?
5. (a) Combien de secondes peut-on écrire sur 32 bits ?
(b) Combien de secondes se sont écoulées depuis le premier janvier 1970 minuit ?

■

Exercice 2 1. Donner les représentations en binaire des réels, puis leur représentation au format IEE754 simple précision :
(a) 74,25 (b) -123,75 (c) 6,125
2. Convertir $0,7$ en binaire et l'exprimer en virgule flottante simple précision.
3. Les expressions `8.5/2.5`, `int(8.5)/int(2.5)` et `int(8.5/2.5)` sont-elles équivalentes ?
4. Combien de nombres dans $[1, 2[$ peut-on coder en simple précision ? Et en double précision ?
5. Que renvoie `0.1 + 0.2 > 0.3` ? Pourquoi ?

■

Exercice 3 1. Écrire une fonction récursive `binaire_rec(n:int) -> str` qui renvoie une chaîne de caractères représentant `n` en base 2. On supposera que `n` est positif.
2. Écrire une fonction `complement(n:int, N:int) -> str` qui renvoie le codage de `n` en complément à 2 sur `N` bits.

■