Chapitre IV
Représentations des nombres

Table des matières

Dans ce chapitre, nous allons voir comment on peut représenter les nombres en mémoire.

Partie A

Représentation des entiers naturels

1. Écriture d'un entier naturels en base b

a. Introduction

De nos jours, quand on pense à un nombre, on pense à 112, 3422, 56 : bref, on entend et on voit un nombre comme une suite de chiffres (symboles) parmi $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$. En fait, les exemples précédents ne sont pas des nombres proprement dits, mais une *représentation* de nombres - plus précisément, **la représentation en base 10** de nombres !
Pour illustrer ce propos, si je dessine une vache, et bien, ce n'est pas une vache ! Simplement la représentation d'une vache. Nous sommes tellement habitués à manipuler des nombres sous la forme précédente qu'on pense à tort que, par exemple, 18 est un nombre alors qu'il s'agit d'une representation du nombre ... ... arf ! Et oui, pour parler ou désigner un nombre, il faut le représenter, car un nombre, en fait, ce n'est pas "réel" comme une vache, c'est un concept, très naturel certes, mais un concept (qui sert à comparer des quantités : "je vois deux vaches et deux chèvres, il y en a autant"). D'où cette confusion entre nombre et représentation d'un nombre.
Mais là où on peut se convaincre que cette distinction a du sens, c'est qu'on peut inventer de nombreuses façons (une infinité !) de représenter un même nombre : si on prend le nombre que nous désignons communément (à notre époque du moins) par 12, on peut le représenter de la façon suivante :

| | | | | | | | | | | | |

Façon qui est d'ailleurs sûrement plus universelle que 12 : en effet, une personne qui n'a jamais appris notre numération en base dix aura beaucoup de mal à déchiffrer (c'est le cas de le dire) l'écriture 12 alors que la précédente est tout de même bien plus limpide. Cette représentation s'appelle la représentation *unaire* - on peut bien-sûr utiliser n'importe quel symbole à la place du bâton.
Alors pourquoi l'homme d'aujourd'hui se complique la vie et n'utilise pas la représentation unaire au quotidien ? Réponse en exercice : *écrire le nombre (oui, oui, représenté par)* 16522 *en représentation unaire.*

Il existe de nombreuses façons exotiques de représenter des nombres, mais dans ce cours, nous allons aborder seulement des représentations cousines de notre représentation en base dix : *les représentations en base b* où b est un entier plus grand que 2 (ou | |). C'est "simple", comme on va le voir : au lieu de faire des tas de puissances de dix, on fera des tas de puissances de b !

Malgré ce beau discours sur la distinction entre nombre et représentation d'un nombre, dans la suite, nous reprendrons notre bonne vieille habitude de dire "nombre" même quand il s'agira d'une représentation de celui-ci... mais au moins, nous en serons conscients !

b. Définitions

Définition 1.

Soit $|$ un symbole fixé. Pour n un entier naturel, on appelle **representation unaire** de n l'écriture :

$$n = \underbrace{|\,|\,|\ \ldots\ |\,|}_{n \text{ termes}}$$

Exemple 1.

- La représentation unaire du nombre quatre est $||||$.
- La représentation unaire de l'unité est $|$.
- La représentation unaire du nombre nul est l'absence de symbole.

Lemme 1.

Soit $b \in \mathbb{N}$ tel que $b \geq 2$ et $n \in \mathbb{N}$. On a :

$$\sum_{k=0}^{n-1}(b-1)b^k = b^n - 1.$$

Démonstration.

On a, par télescopage :

$$\begin{aligned}\sum_{k=0}^{n-1}(b-1)b^k &= \sum_{k=0}^{n-1}(b^{k+1} - b^k)\\ &= b^{n-1+1} - b^0\\ \sum_{k=0}^{n-1}(b-1)b^k &= b^n - 1\end{aligned}$$

□

Lemme 2.

Soit $b \in \mathbb{N}$ tel que $b \geq 2$. Pour tout $N \in \mathbb{N}$, il existe une unique suite $(a_k)_{k\in\mathbb{N}}$ à valeurs dans $[\![0, b-1]\!]$ et stationnaire en 0 telle que :

$$N = \sum_{k=0}^{+\infty} a_k b^k \quad \text{(somme finie)}$$

Démonstration.

> On présente ici une démonstration de ce résultat qui ne "spoilera" pas l'algorithme naturel d'expression d'un entier dans une base que nous verrons plus tard. On peut tout de même produire simplement un algorithme récursif à partir de cette démonstration.

Existence :

Soit $N \in \mathbb{N}$. On note $\mathcal{P}(N)$ la propriété "il existe une suite $(a_k)_{k\in\mathbb{N}}$ à valeurs dans $[\![0, b-1]\!]$ et stationnaire en 0 telle que $N = \sum_{k=0}^{+\infty} a_k b^k$".

Montrons par récurrence sur $\mathbb{N}$ que, pour tout $N \in \mathbb{N}$, $\mathcal{P}(N)$ est vraie.

- **Initialisation :** Si on considère la suite nulle $(0)_{k\in\mathbb{N}}$, celle-ci est à valeurs dans $[\![0, b-1]\!]$, stationnaire en 0 (à partir du rang 0) et on a :

$$\sum_{k=0}^{+\infty} 0.b^k = 0$$

  Donc $\mathcal{P}(0)$ est vraie.

- **Hérédité :** Soit $N \in \mathbb{N}$. On suppose la propriété $\mathcal{P}(N)$ vraie. Alors il existe une suite $(a_k)_{k\in\mathbb{N}}$ à valeurs dans $[\![0, b-1]\!]$ et stationnaire en 0 telle que $N = \sum_{k=0}^{+\infty} a_k b^k$.

  Montrons que $\mathcal{P}(N+1)$ est vraie.
  On considère $I = \{k \in \mathbb{N} \mid a_k < b-1\}$. Alors I est un sous-ensemble non vide (car la suite est stationnaire en 0) de $\mathbb{N}$: il possède donc un plus petit élément que l'on note k_0. On remarque alors que, pour tout entier naturel $k < k_0$, $a_k = b-1$ (ce qui peut être "vide" si $k_0 = 0$).
  On note, pour $k \in \mathbb{N}$:

$$a'_k = \begin{cases} 0 & \text{si } k < k_0 \text{ (si } k_0 = 0 \text{ on ne fait rien !)} \\ a_k + 1 & \text{si } k = k_0 \\ a_k & \text{si } k > k_0 \end{cases}$$

  Alors $(a'_k)_{k\in\mathbb{N}}$ est une suite :
  — à valeurs dans $[\![0, b-1]\!]$ car $(a_k)_{k\in\mathbb{N}}$ l'est et par définition, $a_{k_0} < b-1$ donc $a'_{k_0} = a_{k_0} + 1 \leq b-1$.
  — stationnaire en 0 car $(a_k)_{k\in\mathbb{N}}$ l'est - si on note n le plus petit rang à partir duquel $(a_k)_{k\in\mathbb{N}}$ est stationnaire en 0, celui de $(a'_k)_{k\in\mathbb{N}}$ est égal à n si $k_0 \leq n$ et $n+1$ si $k_0 = n+1$.

D'après le lemme 1, on a $b^{k_0} = 1 + \sum_{k=0}^{k_0-1}(b-1)b^k = 1 + \sum_{k=0}^{k_0-1} a_k b^k$ et ainsi :

$$\begin{aligned}
\sum_{k=0}^{+\infty} a'_k b^k &= \left(\sum_{k=0}^{k_0-1} 0.b^k\right) + (a_{k_0}+1)b^{k_0} + \sum_{k=k_0+1}^{+\infty} a_k b^k \\
&= b^{k_0} + a_{k_0}b^{k_0} + \sum_{k=k_0+1}^{+\infty} a_k b^k \\
&= \left(1 + \sum_{k=0}^{k_0-1} a_k b^k\right) + a_{k_0}b^{k_0} + \sum_{k=k_0+1}^{+\infty} a_k b^k \\
&= 1 + \sum_{k=0}^{+\infty} a_k b^k \\
&= 1 + N \text{ par hypothèse de récurrence} \\
\sum_{k=0}^{+\infty} a'_k b^k &= N+1
\end{aligned}$$

Ce qui prouve l'existence de l'écriture annoncée pour $N+1$.

□

Voici une définition précise de l'écriture en base b d'un nombre. Elle peut paraître compliquée de prime abord ; les exemples que l'on verra ensuite nous permettront de mieux appréhender cette définition.

Définition 2.

Soit $b \in \mathbb{N}$ tel que $b \geq 2$ et $S = \{s^{(0)}, ..., s^{(b-1)}\}$ un ensemble fini de cardinal b.

— Soit a un entier de $[\![0, b-1]\!]$. On dit que $s^{(a)}$ est le *symbole* ou *chiffre* associé à a.

— Soit N un entier naturel où $N = \sum_{k=0}^{n} a_k b^k$ est sa décomposition du lemme 2 et n un entier tel que, pour tout $k > n$, a_k est nul.
On appelle **écriture ou représentation en base b** de N la notation :

$$N = \overline{s_n s_{n-1} \dots s_1 s_0}^b \quad \text{ou encore } N = (s_n s_{n-1} \dots s_1 s_0)_b$$

où, pour tout $k \in [\![0, n]\!]$, $s_k = s^{(a_k)}$ est le symbole le symbole associé à a_k.

Remarque 1.

— Notre façon usuelle d'écrire des nombres, comme 1678 correspond à la base dix.

— Du fait de l'unicité de l'écriture $N = \sum_{k=0}^{+\infty} a_k b^k$ du lemme 2, la représentation de N en base b est unique à des symboles correspondants à zéros initiaux près. Par exemple, en base dix, on a $0001678 = 1678 = 01678$. L'écriture n'est donc pas unique sauf si on ne

tient pas compte des potentiels zéros écrits à gauche.

c. Exemple

Oublions un instant que nous connaissons la base dix et essayons de voir sur un exemple, comment obtenir la représentation en base dix d'un nombre écrit en unaire :

Exemple 2.

Considérons le nombre n représenté en unaire par :

$$||||||||||||||||||||$$

Soit b l'entier représenté en unaire par $||||||||||$ i.e. en bon français, dix. Cherchons la représentation en base b de ce nombre où l'ensemble S des chiffres est (indiqué ici dans l'ordre) $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ c'est-à-dire que :

- le symbole 0 correspond au nombre nul représenté par l'absence de bâton en unaire ;
- le symbole 1 correspond au nombre représenté par $|$ en unaire ;
- le symbole 2 correspond au nombre représenté par $||$ en unaire ;
- etc ...
- le symbole 9 correspond au nombre représenté par $|||||||||$ en unaire ;

On va regrouper notre nombre n par paquets de b et regarder ce qui reste :

$$n = |||||||||| \quad |||||||||| \quad |||$$

On a donc $||$ paquets de b et il reste $|||$ donc

$$n = ||\times b^1 + |||\times b^0.$$

Comme $||$ et $|||$ sont strictement inférieurs à b, la décomposition précédente de n est bien celle du lemme 2. De plus, le symbole correspondant à $||$ est 2 et le symbole correspondant à $|||$ est 3.

Ainsi, la représentation en base b (dix) de n est :

$$\overline{23}^b$$

Voilà, à partir de maintenant, et dans toute la suite, on écrira nos nombres avec notre écriture en base 10 usuelle sans le préciser et on écrira par exemple 18 au lieu de $\overline{18}^{10}$. On réservera cette barre pour les autres bases b avec $b \neq 10$.

d. D'une base b vers la base 10

Exemple 3.

Voyons comment un nombre représenté dans une base donnée s'écrit en base 10 :

base 4 On considère l'ensemble des symboles - l'ensemble des chiffres en base 4,

$$S = \{0, 1, 2, 3\}$$

correspondant dans l'ordre aux nombres entre 0 et 3. Ainsi, le nombre écrit $\overline{1032}^4$ corres-

pond à 78 en base 10, en effet :

$$\overline{1032}^4 = 1 \times 4^3 + 0 \times 4^2 + 3 \times 4 + 2 = 78.$$

base 16 On considère l'ensemble des symboles - l'ensemble des chiffres en base 16,

$$S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f\}$$

correspondant dans l'ordre aux nombres entre 0 et 15. Ainsi, le nombre écrit $\overline{b2f}^{16}$ correspond à 2607 en base 10, en effet :

$$\overline{b2f}^{16} = \underbrace{11}_{b} \times 16^2 + 2 \times 16 + \underbrace{15}_{f} = 2607.$$

base 2 On considère l'ensemble des symboles - l'ensemble des chiffres en base 2,

$$S = \{0, 1\}$$

correspondant dans l'ordre aux nombres 0 et 1. Ainsi, le nombre écrit $\overline{1010}^2$ correspond à 10 en base 10, en effet :

$$\overline{1010}^2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 0 = 10.$$

Les bases les plus couramment utilisées en informatique sont les bases 2 et 16 (outre bien-sûr la base 10). Elles ont donc chacune un nom :

Vocabulaire :

- L'écriture en base 2 est appelée l'écriture en **binaire** ;
- L'écriture en base 16 est appelée l'écriture en **hexadécimale** ;

Exercice 1.

À la main (on a droit à la calculatrice tout de même), écrire en base 10 les nombres représentés dans les bases b indiquées :

1. $\overline{1234}^5$;
2. $\overline{111}^2$;
3. $\overline{100001}^8$;
4. $\overline{abcd}^{14}$;
5. $\overline{fff}^{16}$ et $\overline{1000}^{16}$.

Correction.

1. $\overline{1234}^5 = 194$;
2. $\overline{111}^2 = 7$;

3. $\overline{100001}^{8} = 32769$;
4. $\overline{abcd}^{14} = 29777$;
5. $\overline{fff}^{16} = 4095$ et $\overline{1000}^{16} = 4096$

Exercice 2.

1. Écrire une fonction `base2vers10(k:str) -> int` qui prend en argument une chaîne `k` correspondant à la représentation en base 2 d'un nombre et qui renvoie sa représentation en base 10.
   La fontion `int` permet de convertir une chaîne de caractères numériques en nombre (en base 10) ; par exemple `int('156')` renvoie `156`.

2. On considère l'ensemble S des 36 chiffres de la base 36 i.e. $S = \{0, 1, ..., 9, a, b, c, ...y, z\}$, correspondants, dans l'ordre, aux nombres de 0 à 35.
   Définir un dictionnaire `S` dont les clefs sont les éléments de S sous forme de chaînes de caractères et les valeurs associées sont les nombres correspondants.
   Par exemple :

   ```
   >>>S['4']
   4
   >>>S['f']
   15
   ```

   Pour plus de rapidité, on utilisera la fonction `chr(i:int) -> str` qui *renvoie la chaîne représentant un caractère dont le code de caractère Unicode est le nombre entier* `i` (documentation Python). Pour simplifier, la fonction `chr` associe à un entier un caractère du clavier (et bien d'autres !), par exemple `chr(100)` renvoie `'d'`. Utilisez le fait que les chaînes de `'a'` à `'z'` sont associés aux nombres de 97 à 122.

3. Écrire une fonction `basebvers10(k:str,b:int) -> int` qui prend en argument une chaîne `k` correspondant à la représentation en base `b` d'un nombre et qui renvoie sa représentation en base 10. On s'assurera via un `assert` que `b` est compris entre 2 et 36 et on utilisera le dictionnaire `S` défini à la question précédente !

On pourra écrire deux versions (itérative et récurive) de `base2vers10` et de `basebvers10`

Correction.

1.

```python
## Version itératives
#version gourmande en puissances de 2 !
def base2vers10(k:str) -> int:
    N=0
    l=len(k)
    for i in range(len(k)):
        N+=int(k[l-1-i])*2**i
    return N

#version qui l'est beaucoup moins :
def base2vers10(k:str) -> int:
    N=0
    l=len(k)
    p=1
    for i in range(len(k)):
        N+=int(k[l-1-i])*p
        p*=2
    return N

##Version récursive
def base2vers10(k:str) -> int:
    l=len(k)
    if l==1:
        return int(k[0])
    else:
        return int(k[-1])+base2vers10(k[:-1])*2
```

2.

```python
S={}
for i in range(10):
    S[str(i)]=i
for i in range(97,123):
    S[chr(i)]=i-87
```

3.

```python
def basebvers10(k:str,b:int) -> int:
    assert b>=2 and b<=36
    N=0
    l=len(k)
    p=1
    for i in range(len(k)):
        N+=S[k[l-1-i]]*p
        p*=b
    return N
```

Remarque 2.

La fonction native `int(k:str, b:int) -> int` de Python a exactement le comportement de la fonction `basebvers10(k:str,b:int) -> int` que nous venons de coder !

e. De la base 10 vers une base b

Exemple 4.

Considérons le nombre $n = 301$. Voyons comment il s'écrit dans différentes bases :

base 4 On a :

$$301 = 1 \times 4^4 + 0 \times 4^3 + 2 \times 4^2 + 3 \times 4 + 1$$

donc son écriture en base 4 est :

$$301 = \overline{10231}^4$$

base 16 On a :

$$301 = 1 \times 16^2 + 2 \times 16 + \underbrace{13}_{d}$$

donc son écriture en base 16 est :

$$301 = \overline{12d}^{16}$$

base 2 On a :

$$301 = 1 \times 2^8 + 0 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2 + 1$$

donc son écriture en base 2 est :

$$301 = \overline{100101101}^2$$

Exercice 3.

À la main, écrire les nombres suivants dans les bases indiquées :

1. 25 en base 3 et en base 2 ;
2. 256 et 255 en base 16 ;
3. 230 et 231 en base 11.

Correction.

1. $25 = \overline{221}^2$ et $25 = \overline{11001}^2$;
2. $256 = \overline{100}^{16}$ et $255 = \overline{ff}^{16}$;
3. $230 = \overline{19a}^{11}$ et $230 = \overline{1a0}^{11}$.

Exercice 4.

Prenons un nombre entier naturel n. Pour fixer les idées, prenons $n = 12345$ (écrit en base 10).

1. Comment récupérer le chiffre des unités de n en utilisant une division euclidienne ?
2. Une idée pour récupérer le chiffres des dizaines de n, est de le transformer en $n' = 1234$ et de récupérer, comme à la question précédente, le chiffre des unités de n'. Quelle est cette transformation de n vers n' ?
3. Proposer une façon de récupérer les chiffres des centaines puis des millers de n en s'inspirant de ce qui précéde.
4. Proposer un algorithme puis le programmer via une fonction `chiffres(n:int) -> str` qui prend en argument un entier naturel `n` (écrit naturellement en base 10) et qui renvoie ce même nombre sous forme de chaîne de caractères en utilisant et développant les idées des questions précédentes. On utilisera de nouveau la fonction `str` pour convertir les chiffres récupérés (qui seront des `int` en Python) en chaînes de caractères.
   Voici un exemple de comportement attendu :

   ```
   >>>chiffres(12345)
   '12345'
   ```

   Il n'est bien-sûr pas souhaitable de faire directement `str(n)` qui donne bien le résultat attendu mais qui nous prive totalement de la compréhension de l'algorithme.

   On pourra écrire deux versions (itérative et récurive) de cet algorithme

Correction.

1. Le chiffre des unités de n est obtenu avec son reste dans sa division euclidienne par 10 i.e. `n%10` en Python.
2. Pour passer de n à n', on fait le quotient de la division euclidienne de n par 10 i.e. `n//10` en Python.
3. En reprenant le $n' = n//10$ de la question précédente ; on fait $n'' = n'//10$ puis le chiffre des centaines de n est $n''\%10$.
   De même, on fait $n''' = n''//10$ puis $n'''\%10$ pour obtenir le chiffre des milliers.
4.

```python
#Version itérative
def chiffres(n:int) -> str:
    N=n
    n_str = ''
    while N>0:
        chiffre = N%10
        n_str = str(chiffre)+n_str
        N=N//10
    return n_str

#Version récursive
def chiffres(n:int) -> str:
    if n<10:
        return str(n)
    return chiffres(n//10)+str(n%10)
```

Exercice 5. *Récupération des chiffres en base b d'un nombre*

On a fait le plus dur avec l'exercice précédent ! Pour obtenir les chiffres en base b d'un nombre (pour nous, écrit en base 10), il suffit d'utiliser le même principe que pour la récupération des chiffres en base 10 en remplaçant 10 par ... suspens ... b !

1. On considère l'ensemble S des 36 chiffres de la base 36 i.e. $S = \{0, 1, ..., 9, a, b, c, ...y, z\}$, correspondants, dans l'ordre, aux nombres de 0 à 35.
   Définir une liste `C` dont les éléments sont ceux de S dans l'ordre indiqué et sous forme de chaîne de caractères. Par exemple :

   ```
   >>>C[3]
   '3'
   >>>C[12]
   'c'
   ```

   Pour plus de rapidité, on utilisera de nouveau la fonction `chr`

2. Écrire une fonction `base10versb(n:int,b:int) -> str` qui prend en argument un entier naturel `n` (écrit naturellement en base 10) et qui renvoie la représentation en base b de ce nombre sous forme de chaîne de caractères. On prendra de nouveau b compris entre 2 et 36.

   Indication : commencer par l'écrire en supposant $b \leq 10$ puis faire le cas général $b \leq 36$ en utilisant le liste `C` de la question précédente

3. Écrire une fonction `baseBversb(B:int,n:str,b:int) -> str` qui prend en argument une chaîne de caractères `n` représentant un nombre en base B et qui renvoie la représentation en base b de ce nombre sous forme de chaîne de caractères. On prendra B et b compris entre 2 et 36.

   Indication : on passera par la base 10 ! On pourra ainsi utiliser l'exercice 2 et la question 2 de cet exercice !

Correction.

1.

```
C=[str(i) for i in range(10)]+[chr(i) for i in range(97,123)]
```

2.

```
## cas b<=10
#Itératif
def base10versb(n:int, b:int) -> str:
    assert b>=2 and b<=10
    N=n
    n_str = ''
    while N>0:
        chiffre = N%b
        n_str = str(chiffre)+n_str
        N=N//b
    return n_str

#Récursif
def base10versb(n:int, b:int) -> str:
    assert b>=2 and b<=10
    if n<b:
        return str(n)
    return chiffres(n//b)+str(n%b)

## cas général b<=36
C=[str(i) for i in range(10)]+[chr(i) for i in range(97,123)]

#Itératif
def base10versb(n:int, b:int) -> str:
    assert b>=2 and b<=36
    N=n
    n_str = ''
    while N>0:
        chiffre = N%b
        n_str = C[chiffre]+n_str
        N=N//b
    return n_str

#Récursif
def base10versb(n:int, b:int) -> str:
    assert b>=2 and b<=36
    if n<b:
        return C[n]
    return chiffres(n//b)+C[n%b]
```

3.

```python
C=[str(i) for i in range(10)]+[chr(i) for i in range(97,123)]

S={}
for i in range(10):
    S[str(i)]=i
for i in range(97,123):
    S[chr(i)]=i-87

def baseBversb(B:int,n:str,b:int) -> str:
    return base10versb(basebvers10(n,B), b)
```

f. Représentation binaire et hexadécimale en Python

Comme nous manipulons la base 10 au quotidien, le langage Python nous propose nativement d'utiliser cette base pour communiquer : lorsqu'on entre `213534` dans la console, Python le comprend bien comme un nombre en base 10. Mais comme on le verra dans la partie suivante, il bien plus naturel pour un ordinateur de traiter les informations en la base 2, appelée également représentation **binaire**. Il peut ainsi être utile de pouvoir utiliser la représentation binaire pour communiquer avec l'ordinateur, et ça tombe bien, le langage Python permet de le faire via la syntaxe suivante :

```
>>> 0b1100110111
823
>>> bin(823) #Attention retour de type str !
'0b1100110111'
```

Le problème de l'écriture en binaire d'un nombre est que celle-ci devient vite longue si le nombre est grand - même si cela reste plus économique que la base unaire ! Ainsi, les informaticiens ont souvent recours à la base 16 - appelée base **hexadécimale** pour réduire cette taille. Pourquoi la base 16 et pas notre bonne vieille base 10 ? Comme il s'agit d'une puissance de 2, la conversion avec la base 2 est bien plus simple qu'avec la base 10. En base 16, il faut un symbole pour représenter un entier de 0 à 15 donc tout nombre en base 2 de 4 chiffres est représenté par un seul symbole en base 16. Il suffit donc de découper une représentation en binaire en blocs de 4 chiffres (quitte à rajouter des zéros au début) et de convertir chacun de ces blocs en un symbole hexadécimal pour obtenir sa représentation en base 16.

Table de conversion en hexadécimal d'un "bloc" de 4 chiffres en binaire :

$\overline{0000}^2 = \overline{0}^{16}$	$\overline{0001}^2 = \overline{1}^{16}$	$\overline{0010}^2 = \overline{2}^{16}$	$\overline{0011}^2 = \overline{3}^{16}$
$\overline{0100}^2 = \overline{4}^{16}$	$\overline{0101}^2 = \overline{5}^{16}$	$\overline{0110}^2 = \overline{6}^{16}$	$\overline{0111}^2 = \overline{7}^{16}$
$\overline{1000}^2 = \overline{8}^{16}$	$\overline{1001}^2 = \overline{9}^{16}$	$\overline{1010}^2 = \overline{a}^{16}$	$\overline{1011}^2 = \overline{b}^{16}$
$\overline{1100}^2 = \overline{c}^{16}$	$\overline{1101}^2 = \overline{d}^{16}$	$\overline{1110}^2 = \overline{e}^{16}$	$\overline{1111}^2 = \overline{f}^{16}$

Exemple 5.

Le nombre en binaire $\overline{1010101111110110001}^{2}$ est représenté en héxadécimal par $\overline{2afb1}^{16}$.
En effet, on réalise la conversion par blocs de 4 chiffres, en commençant le regroupement en blocs à droite et en rajoutant des 0 si nécessaire dans le dernier bloc à gauche :

$$\overline{\underbrace{0010}_{\overline{2}^{16}}\ \underbrace{1010}_{\overline{a}^{16}}\ \underbrace{1111}_{\overline{f}^{16}}\ \underbrace{1011}_{\overline{b}^{16}}\ \underbrace{0001}_{\overline{1}^{16}}}^{2} = \overline{2afb1}^{16}$$

Exercice 6.

1. Définir un dictionnaire `bloc_bin_hex` dont les clés sont tous les blocs de 4 chiffres en binaire (de type `str`) et les valeurs correspondantes sont leurs conversions respectives en hexadécimal (encore de type `str`). Par exemple, on aura :

```
>>>bloc_bin_hex['1100']
'c'
```

2. Écrire une fonction `bin_vers_hex(n:str) -> str` qui prend en argument une chaîne de caractères `n` représentant un nombre en binaire et qui renvoie sa représentation en hexadécimal sous forme de chaîne de caractères. On utilisera pour cela la technique de conversion par blocs de 4 en utilisant le dictionnaire `bloc_bin_hex` de la question précédente et en n'oubliant pas de compléter par des zéros au début !

Comme pour le binaire, Python "sait parler" en hexadecimal ; il suffit d'indiquer cette fois-ci le préfixe `0x` :

```
>>> 0xa2ff1b
10682139
>>> hex(10682139) #Attention retour de type str !
'0xa2ff1b'
```

2. Représentation en mémoire des entiers naturels

Pour la mémoire vive et le processeur d'un ordinateur, les données et donc également les nombres entiers naturels peuvent être vu comme des successions de zéros et de uns. En effet, dans un transistor - le processeur étant composé de millions voire milliards d'entre eux, la représentation du zéro et du un est basé sur la répartition des charges négatives et positives en son sein.
Un ordinateur manipule donc les informations codée en binaire : des suites de **bits** valant zéro ou un. Et il est donc naturel d'utiliser la représentation des nombres en binaire pour un ordinateur.

La plupart des processeurs actuels manipulent les données par paquets de 64 bits eux-mêmes regroupés en paquets de 8 bits dont le nom ne vous est pas étranger :

Définition 3. *Octet*

On appelle **octet** une succession de 8 bits.

Ainsi, un processeur 64 bits travaille avec des paquets de 8 octets (faites le compte : $8 \times 8 = 64$!).

En général, les entiers naturels sont utilisés pour représenter les adresses en mémoire. Ces entiers sont codés en binaire et comme la mémoire d'un ordinateur est finie, on doit se limiter à une taille n fixe de bits pour représenter un entier naturel. Ainsi, ce codage sur n bits permet de coder tous les entiers naturels compris entre 0 et $2^n - 1$.

Exemple 6.

— Un octet permet de coder un entier naturel compris entre

$$0 = \overline{00000000}^2 = \overline{00}^{16} \quad \text{et} \quad 2^8 - 1 = 255 = \overline{11111111}^2 = \overline{ff}^{16}$$

Remarque : les couleurs d'une page web sont souvent représentées par trois nombres entiers chacun codé sur un octet et qui représentent ses composantes RVB (Rouge-Vert-Bleu en synthèse additive des couleurs).

— Avec 64 bits, on code les entiers naturels compris entre :

$$0 = \overline{0000000000000000}^{16} \quad \text{et} \quad 2^{64} - 1 = \overline{ffffffffffffffff}^{16}$$

Exemple 7. *Représentations particulières sur n-bits*

On considère un codage des entiers naturels sur n bits. Alors :

— si $p < n$, l'entier 2^p est représenté par $\underbrace{0 \ldots 0}_{n-p-1 \text{ bits}}\ 1\ \underbrace{0 \ldots 0}_{p \text{ bits}}$.

Par exemple, l'entier 2^5 est codé sur un octet par :

0	0	1	0	0	0	0	0

— si $p \leq n$, l'entier $2^p - 1$ est représenté par $\underbrace{0 \ldots 0}_{n-p \text{ bits}}\underbrace{1 \ldots 1}_{p \text{ bits}}$.

Par exemple, l'entier $2^7 - 1$ est codé sur un octet par :

0	1	1	1	1	1	1	1

3. Entiers multi-précision

Sur un oridinateur 64 bits, nous avons vu que le plus grand entiers naturel pouvant être représenté est $2^{64} - 1$. Mais si on donne un nombre plus grand à Python, voilà sa réponse :

```
>>>2**64
18446744073709551616
>>>2**70
1180591620717411303424
>>>2**100
1267650600228229401496703205376
>>>2**1000
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931
```

```
983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376
```

Waouh... Python arrive à outrepasser la limitation matérielle et largement en plus ! Comment fait-il ? Voyons cela :

Définition 4. *Entiers multi-précision*

Soit $n \in \mathbb{N}^*$ et $a \in \mathbb{N}$. La représentation dite en multiprécision de a par blocs de n bits est sa représentation en base $b = 2^n$:

$$a = \overline{a_k....a_0}^{2^n}$$

En machine, les chiffres de a sont codés sous forme d'entiers représentés en binaire sur n bits - chaque chiffre étant compris entre 0 et $2^n - 1$ puis stockés dans une liste.

Exemple 8.

Pour représenter le nombre 548 en multiprécision par blocs de 4 bits, on le décompose en base 2^8 :

$$13548 = 52 \times 256 + 236$$

Puis on écrit chacun des chiffres du nombres en binaire sur 8 bits :

$52 \rightsquigarrow$

0	0	1	1	0	1	0	0

$236 \rightsquigarrow$

1	1	1	0	1	1	1	0

Et enfin, on stocke dans une liste ces deux "chiffres" !

— **Avantages :** on n'a pas de limite de représentation des eniters naturels hormis bien-sûr la taille totale de la mémoire disponible.
— **Inconvénients :** Les opérations d'addition/soustration et multiplication sont plus coûteuses en temps car on ne peut plus utiliser l'arithmétique matérielle directement !

Partie B

Représentation des entiers relatifs

Pour représenter un entier relatif en mémoire, on doit tenir compte du signe et on utilise un bit pour l'encoder. Plus précisément, si on veut encoder un entier relatif sur n bits, le premier bit est réservé pour le signe et les $n-1$ suivants pour ... et bien c'est ce que nous allons voir :

1. Représentation en complément à deux

a. Complément à deux des entiers relatifs positifs

Dans ce cas, c'est très simple :

Représentation en complément à deux des entiers relatifs positifs :
Un entier positif a sur n bits est représenté en complément à deux de la façon suivante : le premier bit vaut 0 et les $n-1$ suivants contiennent la représentation binaire de a.

Par exemple, l'entier $a = 107 = \overline{1101011}^2$ est représenté sur un octet par :

0	1	1	0	1	0	1	1
+	⟵ représentation binaire de a ⟶						

Ainsi, le plus grand entier relatif représentable sur un octet est $2^7 - 1 = 127$. De même, sur une architecture 64 bits, le plus grand entier relatif représentable est $2^{63} - 1 = 9\,223\,372\,036\,854\,775\,807$.

Exercice 7.

Donner la représentation des entiers relatifs suivants sur un octet :

$$14 \quad 32 \quad 127 \quad \overline{7b}^{16}$$

Correction.

1. $14 = \overline{1110}^2$ ⇝

0	0	0	0	1	1	1	0

2. $32 = \overline{100000}^2$ ⇝

0	0	1	0	0	0	0	0

3. $127 = \overline{1111111}^2$ ⇝

0	1	1	1	1	1	1	1

4. $\overline{7b}^{16} = \overline{1111011}^2$ ⇝

0	1	1	1	1	0	1	1

b. Complément à deux des entiers relatifs strictement négatifs

Dans ce cas, la représentation n'est pas celle qu'on attendrait de prime abord : en effet, on pourrait calquer le cas des entiers positifs en mettant 1 pour le premier bit et en enchaînant avec la représentation binaire du nombre en valeur absolue. Mais cette approche possède des inconvénients :

- on n'aurait pas unicité de la représentation du zéro ;
- mais surtout, l'algorithme d'addition entre entiers de signes opposés ne serait pas la même que celui pour des entiers de même signe.

Pour pallier ces incovénients, on utilise une représentation des entiers négatifs qui semble moins naturelle mais qui permet d'éviter les deux problèmes précédents : la représentation en *complément à deux*.

Lemme 3.

Soit $n \in \mathbb{N}^*$ et $a \in [\![-2^{n-1}, -1]\!]$. Alors :

$$\overline{1\underbrace{0000\ldots000}_{n-1 \text{ chiffres}}}^2 = 2^{n-1} \leq 2^n + a \leq 2^n - 1 = \overline{\underbrace{1111\ldots1111}_{n \text{ chiffres}}}^2$$

En particulier, la représentation binaire de $2^n + a$ possède exactement n chiffres.

Démonstration.

On a :

$$\begin{array}{lrcl} & -2^{n-1} \leq & a & \leq -1 \\ \text{d'où} & 2^n - 2^{n-1} \leq & 2^n + a & \leq 2^n - 1, \text{ or } 2^n = 2 \times 2^{n-1}, \\ \text{donc} & 2^{n-1} \leq & 2^n + a & \leq 2^n - 1 \end{array}$$

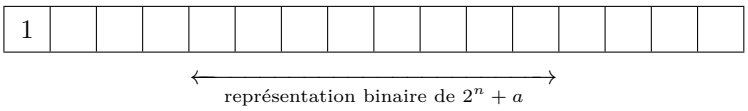
□

Définition 5. *Complément à deux d'un entier strictement négatif*

On appelle **représentation en complément à deux** des entiers strictement négatifs, la représentation suivante (sur n bits) :

un entier a de l'intervalle $[\![-2^{n-1}, -1]\!]$ est représenté en complément à deux sur n bits par la représentation binaire de $2^n + a$. Celle-ci possèdant exactement n chiffres, le premier bit de la représentation en complément à deux de a est un 1.

Représentation en complément à deux de a :



Exemple 9.

Considérons l'entier $a = -107 = -\overline{1101011}^2$ et déterminons sa représentation en complément à deux sur un octet ($n = 8$) - ce qui est possible car $-2^7 \geq a \geq -1$:
On a $2^+a = 256 - 107 = 149 = \overline{10010101}^2$. Ainsi, sa représentation en complement à deux est :

1	0	0	1	0	1	0	1

Proposition 1.

Soit $n \in \mathbb{N} \setminus \{0,1\}$ et $a \in [\![-2^{n-1}, -1]\!]$ de représentation binaire $a = -\overline{a_{n-2}\dots a_0}^2$ où les $a_i \in \{0,1\}$. Alors :

$$2^n + a = \left(\overline{1\, b_{n-2}\dots b_0}^2\right) + \mathbf{1}$$

où, pour tout $i \in [\![1, n-2]\!]$,

$$b_i = 1 - a_i = \begin{cases} 0 & \text{si } a_i = 1 \\ 1 & \text{si } a_i = 0 \end{cases}$$

Démonstration.

On a $a = -\overline{a_{n-2}\dots a_0}^2 = -\sum_{i=0}^{n-2} a_i 2^i$ donc comme $2^n = 1 + \sum_{i=0}^{n-1} 2^i$, on a :

$$2^n + a = 2^{n-1} + \left(\sum_{i=1}^{n}(1-a_i)2^i\right) + 1 = \overline{1\,(1-a_{n-2})\dots(1-a_0)}^2 + 1 = \overline{1\, b_{n-2}\dots b_0}^2 + 1$$

□

Le complement à deux en pratique : En utilisant la proposition précédente, on se rend compte qu'il y a une manière simple d'obtenir la représentation en complément à deux sur n bits à partir de la représentation binaire d'un entier a compris entre -2^{n-1} et -1 :

- On "inverse" chaque bit - 1 devient 0 et réciproquement - de la représentation binaire de $|a|$ sur $n-1$ bits (cette opération est appelé *complément à un*) ;
- puis on ajoute 1 au résultat ;

Exemple 10.

On cherche la représentation en complément à deux de -12 sur un octet :

- Sur 8 bits, la représentation binaire de $|-12| = 12 = \overline{1100}^2$ est :

0	0	0	0	1	1	0	0

On effectue un complément à un en inversant chaque bits :

1	1	1	1	0	0	1	1

- puis on ajoute 1 :

1	1	1	1	0	0	1	1	$+1$	$\rightsquigarrow$	1	1	1	1	0	1	0	0

Donc la représentation en complément à deux de -12 est :

1	1	1	1	0	1	0	0

c. Complément à deux en général

La représentation des entiers relatifs sur n bits que nous avons décrite en deux parties (pour les positifs puis pour les strictement négatifs) est appelé représentation en complément à deux que ce soit pour les négatifs mais aussi pour les positifs.

Définition 6. *Complément à deux d'un entier relatif*

On appelle **représentation en complément à deux** d'un entier relatif, la représentation suivante (sur n bits) :

Pour un entier $a \in [\![-2^{n-1}, 2^{n-1}-1]\!]$:

- Si $a \in [\![0, 2^{n-1}-1]\!]$, le complément à deux sur n bits de a est le représentation binaire de a sur n bits.
- Si $a \in [\![-2^{n-1}, -1]\!]$, le complément à deux sur n bits de a est le représentation binaire de $2^n + a$ sur n bits.

Proposition 2.

Soit $n \geq 2$. Sur n bits, la représentation en complement à deux permet d'encoder de manière unique tous les entiers relatifs compris entre -2^{n-1} et $2^{n-1}-1$.

Exercice 8.

1. Représenter en complément à deux sur un octet les entiers relatifs suivants :

25	127	0	128
-44	-1	-150	-128

2. Déterminer les nombres relatifs, exprimés en base 10, dont les représentations en complement à deux sur un octet sont les suivantes :

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

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

Correction.

1. (a) On a $-2^{8-1} = -128 \leq 25 \leq 127 = 2^{8-1}-1$, donc 25 est bien représentable en complément à deux sur 8 bits. De plus, $25 = \overline{11001}^2$ donc sa représentation en complement à deux sur un octet est :

$$25 \rightsquigarrow \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ \hline \end{array}$$

   (b) On a $-2^{8-1} = -128 \leq 127 \leq 127 = 2^{8-1}-1$, donc 127 est bien représentable en complément à deux sur 8 bits. De plus, $127 = \overline{1111111}^2$ donc sa représentation en

complement à deux sur un octet est :

$$127 \rightsquigarrow \boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}$$

(c) On a $-2^{8-1} = -128 \leq 0 \leq 127 = 2^{8-1} - 1$, donc 0 est bien représentable en complément à deux sur 8 bits. De plus, $0 = \overline{0}^2$ donc sa représentation en complement à deux sur un octet est :

$$0 \rightsquigarrow \boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}$$

(d) On a $128 > 127 = 2^{8-1}-1$ donc 128 n'est pas représentable sur un octet en complement à deux.

(e) On a $-2^{8-1} = -128 \leq -44 \leq 127 = 2^{8-1} - 1$, donc -44 est bien représentable en complément à deux sur 8 bits. De plus, $44 = \overline{101100}^2$ donc

- Sur 8 bits, sa représentation binaire est :

$$\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{0}$$

On effectue un complément à un en inversant chaque bits :

$$\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{1}$$

- puis on ajoute 1 :

$$\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{1} + 1 \rightsquigarrow \boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{0}$$

Donc la représentation en complément à deux de -44 est :

$$\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{0}$$

(f) On a $-2^{8-1} = -128 \leq -1 \leq 127 = 2^{8-1} - 1$, donc -1 est bien représentable en complément à deux sur 8 bits. De plus, $1 = \overline{1}^2$ donc

- Sur 8 bits, sa représentation binaire est :

$$\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{1}$$

On effectue un complément à un en inversant chaque bits :

$$\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{0}$$

- puis on ajoute 1 :

$$\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{0} + 1 \rightsquigarrow \boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}$$

Donc la représentation en complément à deux de -1 est :

$$\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}$$

(g) On a $-150 < -128 > -2^{8-1}$ donc -150 n'est pas représentable sur un octet en complement à deux.

(h) On a $-2^{8-1} = -128 \leq -128 \leq 127 = 2^{8-1} - 1$, donc -128 est bien représentable en complément à deux sur 8 bits. De plus, $128 = \overline{10000000}^2$ donc

- Sur 8 bits, sa représentation binaire est :

1	0	0	0	0	0	0	0

On effectue un complément à un en inversant chaque bits :

0	1	1	1	1	1	1	1

- puis on ajoute 1 :

0	1	1	1	1	1	1	1	$+1$	$\rightsquigarrow$	1	0	0	0	0	0	0	0

Donc la représentation en complément à deux de -128 est :

1	0	0	0	0	0	0	0

2. (a) Comme la représentation de n commence par un 0, il s'agit d'un nombre positif et donc

$$n = \overline{1110100}^2 = 116$$

(b) Comme la représentation de n commence par un 0, il s'agit d'un nombre positif et donc

$$n = \overline{1010101}^2 = 85$$

(c) Comme la représentation de n commence par un 1, il s'agit d'un nombre strictement négatif. On obtient donc sa valeur en binaire de la façon suivante :

— On lui soustrait 1 ce qui donne :

1	1	0	1	0	1	0	0

— Et on "inverse" les bits pour obtenir la représentation binaire de sa valeur absolue, ce qui donne :

0	0	1	0	1	0	1	1

D'où :

$$n = -\overline{101011}^2 = -43$$

(d) Comme la représentation de n commence par un 1, il s'agit d'un nombre strictement négatif. On obtient donc sa valeur en binaire de la façon suivante :

— On lui soustrait 1 ce qui donne :

1	0	1	0	0	1	1	0

— Et on "inverse" les bits pour obtenir la représentation binaire de sa valeur absolue, ce qui donne :

0	1	0	1	1	0	0	1

D'où :

$$n = -\overline{1011001}^2 = -89$$

(e) Comme la représentation de n commence par un 1, il s'agit d'un nombre strictement négatif. On obtient donc sa valeur en binaire de la façon suivante :

— On lui soustrait 1 ce qui donne :

1	1	1	1	0	1	0	1

— Et on "inverse" les bits pour obtenir la représentation binaire de sa valeur absolue, ce qui donne :

0	0	0	0	1	0	1	0

D'où :

$$n = -\overline{1010}^2 = -10$$

(f) Comme la représentation de n commence par un 1, il s'agit d'un nombre strictement négatif. On obtient donc sa valeur en binaire de la façon suivante :

— On lui soustrait 1 ce qui donne : | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |

— Et on "inverse" les bits pour obtenir la représentation binaire de sa valeur absolue, ce qui donne : | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

D'où :

$$n = -\overline{1000000}^2 = -64$$

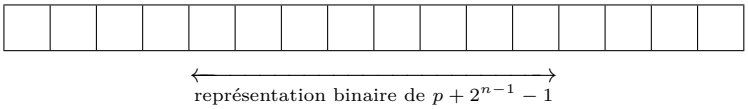
2. Représentation en codage par excès

La représentation en codage par excès que nous allons étudier a le mérite d'être simple : une seule formule est nécessaire pour encoder tous les entiers relatifs d'un intervalle donné. Par contre, par rapport à la représentation en complément à deux, cette simplicité a un désavantage : l'algorithme de d'addition n'est plus aussi simple.
Malgré cela, c'est cette représentation que nous utiliserons dans la suite dans la représentation des nombres flottants.

Définition 7. *Codage binaire des entiers relatifs par excès*

Soit $n \in \mathbb{N}^*$ et p un entier entre $-2^{n-1}+1$ et 2^{n-1}. On appelle **codage binaire par excès** de p la représentation binaire sur n bits du nombre entier naturel $q = p + 2^{n-1} - 1$.

Codage par excès de p :



Exemple 11.

Voici le tableau de codage binaire par excès sur 3 bits des entiers p entre $-3 = -2^2+1$ et $4 = 2^2$:

p (base 10)	$q = p + 2^{n-1} - 1 = p + 3$	codage par excès de p
-3	0	0 0 0
-2	1	0 0 1
-1	2	0 1 0
0	3	0 1 1
1	4	1 0 0
2	5	1 0 1
3	6	1 1 0
4	7	1 1 1

Exercice 9.

1. Représenter en codage par excès sur un octet les entiers relatifs suivants :

$$\begin{array}{cccc} 25 & 128 & 0 & 150 \\ -44 & -1 & -128 & -127 \end{array}$$

2. Déterminer les nombres relatifs, exprimés en base 10, dont les représentations en codage par excès sur un octet sont les suivantes :

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

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

Correction.

1. — Le nombre $p = 25$ est bien compris entre $-127 = -2^{8-1} + 1$ et $128 = 2^{8-1}$ donc il est représentable en codage par excès sur 8 bits, et on a :

$$q = p + 2^{8-1} - 1 = 25 + 127 = 152 = \overline{10011000}^2.$$

Ainsi, la representation de 25 en codage par excès sur un octet est :

$$25 \rightsquigarrow \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ \hline \end{array}$$

— Le nombre $p = 128$ est bien compris entre $-127 = -2^{8-1} + 1$ et $128 = 2^{8-1}$ donc il est représentable en codage par excès sur 8 bits, et on a :

$$q = p + 2^{8-1} - 1 = 128 + 127 = 255 = \overline{11111111}^2.$$

Ainsi, la representation de 128 en codage par excès sur un octet est :

$$128 \rightsquigarrow \begin{array}{|c|c|c|c|c|c|c|c|} \hline 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline \end{array}$$

— Le nombre $p = 0$ est bien compris entre $-127 = -2^{8-1} + 1$ et $128 = 2^{8-1}$ donc il est représentable en codage par excès sur 8 bits, et on a :

$$q = p + 2^{8-1} - 1 = 0 + 127 = 127 = \overline{1111111}^2.$$

Ainsi, la representation de 0 en codage par excès sur un octet est :

$$0 \rightsquigarrow \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ \hline \end{array}$$

— Le nombre $p = 150$ n'est pas compris entre $-127 = -2^{8-1} + 1$ et $128 = 2^{8-1}$ donc il n'est pas représentable en codage par excès sur 8 bits.

— Le nombre $p = -44$ est bien compris entre $-127 = -2^{8-1} + 1$ et $128 = 2^{8-1}$ donc il est représentable en codage par excès sur 8 bits, et on a :

$$q = p + 2^{8-1} - 1 = -44 + 127 = 83 = \overline{1010011}^2.$$

Ainsi, la representation de -44 en codage par excès sur un octet est :

$$-44 \rightsquigarrow \begin{array}{|c|c|c|c|c|c|c|c|} \hline 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ \hline \end{array}$$

— Le nombre $p = -1$ est bien compris entre $-127 = -2^{8-1} + 1$ et $128 = 2^{8-1}$ donc il est représentable en codage par excès sur 8 bits, et on a :

$$q = p + 2^{8-1} - 1 = -1 + 127 = 126 = \overline{1111110}^2.$$

Ainsi, la representation de -1 en codage par excès sur un octet est :

$-1 \rightsquigarrow$ | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |

— Le nombre $p = -128$ n'est pas compris entre $-127 = -2^{8-1} + 1$ et $128 = 2^{8-1}$ donc il n'est pas représentable en codage par excès sur 8 bits.

— Le nombre $p = -127$ est bien compris entre $-127 = -2^{8-1} + 1$ et $128 = 2^{8-1}$ donc il est représentable en codage par excès sur 8 bits, et on a :

$$q = p + 2^{8-1} - 1 = -127 + 127 = 0 = \overline{0}^2.$$

Ainsi, la representation de -127 en codage par excès sur un octet est :

$-127 \rightsquigarrow$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

2. — On a $q = \overline{01110100}^2$ = donc $p = q + 1 - 2^{8-1} = 116 - 127 = -11$. Ainsi, le nombre en base 10 représenté par | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | est -11.

— On a $q = \overline{01010101}^2$ = donc $p = q + 1 - 2^{8-1} = 85 - 127 = -42$. Ainsi, le nombre en base 10 représenté par | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | est -42.

— On a $q = \overline{11010101}^2$ = donc $p = q + 1 - 2^{8-1} = 213 - 127 = 86$. Ainsi, le nombre en base 10 représenté par | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | est 86.

— On a $q = \overline{10100111}^2$ = donc $p = q + 1 - 2^{8-1} = 167 - 127 = 40$. Ainsi, le nombre en base 10 représenté par | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | est 40.

— On a $q = \overline{11110110}^2$ = donc $p = q + 1 - 2^{8-1} = 246 - 127 = 119$. Ainsi, le nombre en base 10 représenté par | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | est 119.

— On a $q = \overline{11000000}^2$ = donc $p = q + 1 - 2^{8-1} = 192 - 127 = 65$. Ainsi, le nombre en base 10 représenté par | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | est 65.

Exercice 10.

Écrire une fonction `codage_exces(n:int) -> None` qui permet d'afficher le tableau de codage par excès sur n bits.

Correction.

```
from math import log
def codage_exces(n:int) -> None:
    N=2**(n-1)
    nb = int(n*log(2)/log(10))+1 # nombre de caractères maximum pour la colonne
         q en décimal
    def cellule(contenu:str,taille_cellule:int) -> str:
        return contenu+' '*(taille_cellule-len(contenu))
    ligne = '-'*(2*nb+n+5)
    print(ligne)
    print('|'+cellule('p',nb+1)+'|'+cellule('q',nb)+'|'+cellule('ex',n)+'|')
    print(ligne)
    for q in range(2*N):
        print('|'+cellule(str(q-N+1),nb+1)+'|'+cellule(str(q),nb)+'|'+cellule(
            bin(q)[2:],n)+'|')
        print(ligne)
```

Partie C

Représentation des nombres flottants

La mémoire d'un ordinateur étant finie, il est clair qu'un nombre réel tel que $\sqrt{2}$ ou π ne pourra pas être représenté sur une machine sans en faire une approximation.
Nos ordinateurs manipulant le binaire, on se limite au stockage des nombres rationnels dyadique - un joli mot pour l'équivalent des nombres décimaux mais en base 2 ; bref, des nombres s'écrivant $\frac{a}{2^n}$ avec a entier relatif et n entier naturel. Bien-sûr, seuls un nombre fini d'entre eux seront représentables, toujours par finitude de la machine.
La représentation que nous allons étudier de ces nombres s'appelle la représentation à virgule flottante et est analogue à l'écriture scientifique qu'on utilise très souvent en physique.

1. Écriture binaire d'un nombre dyadique

Définition 8.

On appelle nombre dyadique un nombre réel qui s'écrit sous la forme $\frac{a}{2^n}$ avec $a \in \mathbb{Z}$ et $n \in \mathbb{N}$.

Définition-Proposition 9. *Représentation binaire d'un nombre dyadique*

Soit x un nombre dyadique positif. Alors il existe une unique décomposition (à des zéros près "à chaque bout")

$$x = \sum_{k=-m}^{n} x_k 2^k \quad \text{où } x_k \in \{0,1\}$$

La **représentation binaire** de x est l'écriture :

$$x = \overline{x_n \ldots x_0, x_{-1} \ldots x_{-m}}^2.$$

Démonstration.

Soit x un nombre dyadique positif. Alors il existe $a \in \mathbb{N}$ et $m \in \mathbb{N}^*$ tels que $x = \frac{a}{2^m}$. Par suite, on a $2^b x = a \in \mathbb{N}$, donc, d'après le lemme 2, $x2^m$ admet une unique décomposition :

$$x2^m = \sum_{k=0}^{p} a_k 2^k \text{ où les } a_i \in \{0,1\}.$$

Par suite,

$$x = \sum_{k=0}^{p} a_k 2^{k-m} = \sum_{k=-m}^{p-m} a_{k+m} 2^k$$

En posant, pour tout $k \in [\![-m, p-m]\!]$, $x_k = a_{k+m}$ et si $p - m < 0$, pour tout $k \in [\![p-m+1, 0]\!]$,

$x_k = 0$, on obtient, en notant $n = \max(0, p - m)$:

$$x = \sum_{k=-m}^{n} x_k 2^k$$

Et on a bien les $x_k \in \{0, 1\}$. L'unicité de cette décomposition découle de celle de $x2^m$ donnée par le lemme 2. □

Exemple 12.

Le nombre $x = \overline{10,011}^2$ vaut en base 10 :

$$\begin{aligned} x &= 1 \times 2^1 + 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} \\ &= 2 + 0,25 + 0,125 \\ x &= 2,375 \end{aligned}$$

Exercice 11.

Représenter en base 10 les nombres suivants :

$$\overline{1,1}^2 \quad \overline{101,11}^2 \quad \overline{11,1001}^2$$

Correction.

1. $\overline{1,1}^2 = 1,5$
2. $\overline{101,11}^2 = 5,75$
3. $\overline{11,1001}^2 = 3,5625$

Conversion d'un nombre décimal en base 10 vers le binaire :

1. on recupère l'entier à gauche de la virgule et on le convertit en binaire ;
2. on récupère la partie fractionnaire à droite de la virgule et tant que la partie fractionnaire n'est pas nulle :
   — on la multiplie par 2 ;
   — on récupère et soustrait le bit à gauche de la virgule (donc forcément 0 ou 1, pourquoi ?) ;
3. on met à gauche de la virgule la représentation binaire du 1. et à droite de la virgule, la suite des bits obtenus au 2.

Si le 2. ne se termine pas, c'est que le nombre n'était pas dyadique !

Exemple 13.

Considérons le nombre $x = 5,3125$

1. À gauche de la virgule $5 = \overline{101}^2$;

2. la partie fractionnaire est $0,3125$ et on applique l'algorithme :

$$\begin{array}{rcrcrcrcr} 0,3125 & \times & 2 & = & 0,625 & \rightsquigarrow & \mathbf{0} & + & 0,625 \\ 0,625 & \times & 2 & = & 1,25 & \rightsquigarrow & \mathbf{1} & + & 0,25 \\ 0,25 & \times & 2 & = & 0,5 & \rightsquigarrow & \mathbf{0} & + & 0,5 \\ 0,5 & \times & 2 & = & 1,0 & \rightsquigarrow & \mathbf{1} & + & 0,0 \end{array}$$

3. Ainsi, $5,3125 = \overline{101,0101}^2$.

Exemple 14.

Considérons le nombre $x = 0,1$.

1. À gauche de la virgule $0 = \overline{0}^2$;
2. la partie fractionnaire est $0,1$ et on applique l'algorithme :

$$\begin{array}{rcrcrcrcr} 0,1 & \times & 2 & = & 0,2 & \rightsquigarrow & \mathbf{0} & + & 0,2 \\ 0,2 & \times & 2 & = & 0,4 & \rightsquigarrow & \mathbf{0} & + & 0,4 \\ 0,4 & \times & 2 & = & 0,8 & \rightsquigarrow & \mathbf{0} & + & 0,8 \\ 0,8 & \times & 2 & = & 1,6 & \rightsquigarrow & \mathbf{1} & + & 0,6 \\ 0,6 & \times & 2 & = & 1,2 & \rightsquigarrow & \mathbf{1} & + & 0,2 \end{array}$$

et c'est reparti !

3. Ainsi, $0,1 = \overline{0,0001100110011001...}^2$, donc $0,1$ n'est pas un nombre dyadique !

Exercice 12.

Représenter en binaire les nombres suivants :

$$6,625 \quad 25,192626953125 \quad 0,15$$

Correction.

1. $6,625 = \overline{110,101}^2$
2. $25,255 = \overline{11001,001100010101}^2$
3. $0,15 = \overline{0,001001100110011...}^2$

Exercice 13.

Écrire une fonction `dya_10_vers_bin(x:float) -> str` qui prend en argument un nombre dyadique `x` appartenant à l'intervalle $[0,1[$ (pour simplifer et ne traiter que la partie fractionnaire) implémente l'algorithme précédent.
Par exemple,

```
>>> dya_10_vers_bin(0.625)
'0,101'
```

Correction.

```python
#Version itérative
def dya_10_vers_bin(x:float) -> str:
    y=x
    binaire = '0,'
    while y!=0:
        y*=2
        bit = int(y)
        y -= bit
        binaire += str(bit)
    return binaire

#Version récursive
def dya_10_vers_bin_rec(x:float) -> str:
    def rec(x):
        if x==0:
            return ''
        y=x*2
        bit = int(y)
        return str(bit)+rec(y-bit)
    return '0,'+rec(x)
```

Lemme 4.

Soit $x = \frac{a}{2^n}$ un nombre dyadique strictement positif. Alors il existe un unique couple (m, p) où $m \in [0, 1[$ est un nombre dyadique et $p \in \mathbb{Z}$ tel que :

$$x = (1 + m).2^p$$

Démonstration.

Comme $a \geq 1$ (car a est un entier strictement positif), il existe un unique $k \in \mathbb{N}$ tel que $2^k \leq a < 2^{k+1}$. On pose alors $p = k - n \in \mathbb{Z}$ et $m = \frac{a}{2^k} - 1$. Ainsi, comme $\frac{a}{2^k} \in [1, 2[$, $m \in [0, 1[$ et $m = \frac{a-2^k}{2^k}$ est dyadique.
De plus, on a :

$$(1 + m).2^p = \frac{a}{2^k}.2^{k-n} = \frac{a}{2^n} = x.$$

Pour l'unicité, si $\frac{a'}{2^{n'}}$ est un autre représentant de x, alors on a $a' = a \times 2^{n-n'}$ d'où, par la même

construction avec des "primes", on a $k' = k + n' - n$ et donc :

$$p' = k' - n' = k - n \text{ et } m' = \frac{a'}{2^{k'}} - 1 = \frac{a}{2^k} - 1 = m.$$

Donc le couple (m, p) est unique. □

Définition 10. *Écriture scientifique binaire*

Soit x un nombre dyadique non nul. On appelle **écriture scientique binaire** de x, l'écriture :

$$x = \pm\, \overline{1, M}^2 \times 2^p$$

où

— $\overline{1, M}^2$ est l'écriture binaire du nombre $1 + m$ donné par le lemme précédent. La suite de bits M est appelée la **mantisse** de x.

— p est la puissance donnée par le lemme précédent.

Exemple 15.

Décimal	Écriture sc.	Binaire	Écriture sc. binaire	Mantisse
$58,25$	$+5,825 \times 10^1$	$\overline{111010,01}^2$	$+\overline{1,1101001}^2 \times 2^5$	1101001
-0.0048828125	$-4,8828125 \times 10^{-3}$	$-\overline{0,0000000101}^2$	$-\overline{1,01}^2 \times 2^{-8}$	01
$128,09375$	$+1,2809375 \times 10^2$	$\overline{10000000,00011}^2$	$+\overline{1,000000000011}^2 \times 2^7$	000000000011

Exercice 14.

1. Déterminer l'écriture scientique binaire des nombres suivants en précisant leur mantisse :

$$-22,125 \qquad 0,000244140625$$

2. Déterminer la représentation en base 10 des nombres $x = \pm\, \overline{1, M}^2 \times 2^p$ en écriture scientifique binaire où :
   (a) signe : $+$; mantisse : $M = 01$; puissance : $p = -3$;
   (b) signe : $-$; mantisse : $M = 110111$; puissance : $p = 4$;

Correction.

1. (a) $-22,125 = -\overline{10110,001}^2 = -\overline{1,0110001}^2 \times 2^4$. Mantisse : 0110001.
   (b) $0,000244140625 = \overline{0,000000000001}^2 = +\overline{1,0}^2 \times 2^{-12}$. Mantisse : 0.
2. (a) $x = +\overline{1,01}^2 \times 2^{-3} = \overline{0,00101}^2 = 0,15625$;
   (b) $x = -\overline{1,110111}^2 \times 2^4 = -\overline{11101,11}^2 = -29,75$;

2. Représentation en mémoire des nombres flottants

La représentation en mémoire des nombres en virgule flottante repose directement sur l'écriture scientifique binaire que nous avons étudié au paragraphe précédent. On stocke sur des nombres de bits définis : le signe ; la mantisse et la puissance de l'écriture scientifique binaire.
Ainsi, seul un nombre fini de nombres dyadiques sont représentables de cette façon.

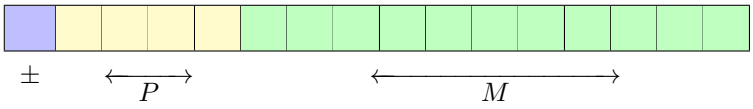
Définition 11. *Nombres à virgule flottante*

Soit $n \in \mathbb{N}^*$ et m, p des entiers tels que $1 + m + p = n$. On dit qu'un nombre dyadique $x = \pm\overline{1,M}^2 \times 2^P$ est un **nombre à virgule flottante sur n bits** ou un **flottant sur n bits** si :

- la mantisse M peut être stockée sur m bits ;
- la puissance P peut être stockée en codage binaire par excès sur p bits.

Dans ce cas, on encode le flottant x sur n bits sous la forme et dans l'ordre suivant :

- le signe sur 1 bit (0 pour $+$, 1 pour $-$) ;
- la puissance P sur p bits en codage binaire par excès ;
- la mantisse M sur m bits en binaire **en complétant <u>à droite</u> par des 0** (et non à gauche comme d'habitude !).



Le choix de m et p pour un n donné dans la définition précédente sont arbitraires et font donc l'objet d'un normalisation : actuellement, la norme IEEE-754 est la plus employée pour le stockage des flottants. Voici deux formats de flottants de cette norme :

Formats simple et double précision de la norme IEEE-754 pour les flottants :

n	format	signe	puissance	mantisse
32 bits	simple précision	1 bit	8 bits	23 bits
64 bits	double précision	1 bit	11 bits	52 bits

Pour Python, les flottants - de type `float` - sont codés en double précision selon la norme IEEE-754.

Exemple 16.

Considérons le nombre $n = -3123,125$ et voyons s'il s'agit d'un nombre à virgule flottante en double précision.
On détermine l'écriture scientifique binaire de n :

$$n = -\overline{110000110011,001}^2 = -\overline{1,10000110011001}^2 \times 2^{11}$$

Alors :

- $P = 11$ est compris entre $-2^{10} + 1$ et 2^{10} donc son codage binaire par excès sera possible

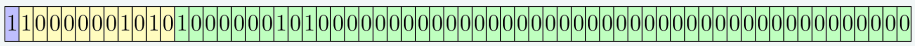
sur 11 bits. Plus précisément :

$$P = 11 \rightsquigarrow Q = P + 2^{10} - 1 = 1034 = \overline{10000001010}^2$$

d'où le codage binaire de P par excès sur 11 bits :

1	0	0	0	0	0	0	1	0	1	0

— $M = 10000110011001$ est de longueur 14 bits $\leq$ 52 bits.
Ainsi, n est représentable en flottant double précision et sa représentation est :

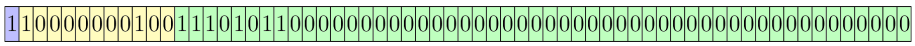


Exercice 15.

1. Donner la représentation en flottant double précision de :

$$0,0703125 \text{ et } -1024.00048828125$$

2. Quel est est le nombre, exprimé en base 10, dont la représentation en flottant double précision est :



Correction.

1. (a) On détermine l'écriture scientifique binaire de $n = 0,0703125$:

$$n = \overline{0,0001001}^2 = +\overline{1,001}^2 \times 2^{-4}$$

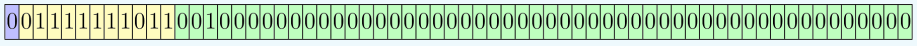
Alors :
— $P = -4$ est compris entre $-2^{10} + 1$ et 2^{10} donc son codage binaire par excès sera possible sur 11 bits. Plus précisément :

$$P = -4 \rightsquigarrow Q = P + 2^{10} - 1 = 1019 = \overline{1111111011}^2$$

d'où le codage binaire de P par excès sur 11 bits :

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

— $M = 001$ est de longueur 3 bits $\leq$ 52 bits.
Ainsi, $0,0703125$ est représentable en flottant double précision et sa représentation est :



(b) On détermine l'écriture scientifique binaire de $n = -1024.00048828125$:

$$n = -\overline{10000000000,00000000001}^2 = -\overline{1,00000000000000000001}^2 \times 2^{10}$$

Alors :

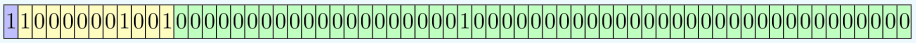
— $P = 10$ est compris entre $-2^{10}+1$ et 2^{10} donc son codage binaire par excès sera possible sur 11 bits. Plus précisément :

$$P = 10 \rightsquigarrow Q = P + 2^{10} - 1 = 1033 = \overline{10000001001}^2$$

d'où le codage binaire de P par excès sur 11 bits :

1	0	0	0	0	0	0	1	0	0	1

— $M = 00000000000000000001$ est de longueur 21 bits $\leq$ 52 bits.
Ainsi, -1024.00048828125 est représentable en flottant double précision et sa représentation est :



2. On cherche $x = \overline{1,M}^2 \times 2^P$.
   — Le signe est encodé par 1 donc il s'agit de $-$.
   — Le codage binaire par excès de P est $Q = \overline{10000000100}^2 = 1028$. Donc $P = Q - 2^{10} + 1 = 5$.
   — La mantisse M est 11101011.
   Ainsi,

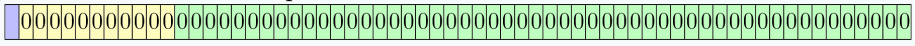
$$x = -\overline{1,11101011}^2 \times 2^5 = -1,91796875 \times 2^5 = -61.375$$

3. Valeurs réservées pour les flottants

A priori, le nombre 0 n'est pas représentable en nombre à virgule flottante : en effet, à cause du "1," précédant la mantisse dans l'écritutre scientifique binaire, tout flottant est non nul !
Pour pallier ce problème, dans la norme IEEE-754, on réserve une "place" parmi les flottants avec la convention qu'il s'agit de 0.
En fait, on a même d'autres flottants dont on va prendre la place ! Voici les exemples les plus importants :

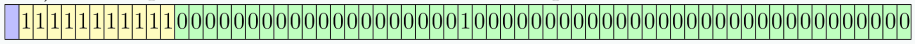
Exemples de flottants double précision réservés :

— **Zéro :** Pour encoder 0, on réserve les flottants dont la puissance (en codage binaire par excès) et la mantisse sont remplies de bit 0.



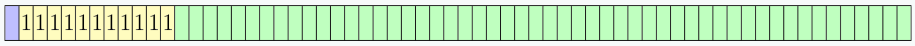
On a donc deux représentations de 0, une positive et une négative selon le bit de signe.
Les nombres réservés dont 0 a pris la place sont $\pm 2^{-1023}$ - en fait les nombres qui sont censés être représentables en flottant les plus proches de zéros.

— **Infinis :** Pour encoder $\pm\infty$, on réserve les flottants dont la puissance (en codage binaire par excès) est remplie de bit 1 et la mantisse est remplie de bit 0.



Avec bien-sûr le bit de signe 0 pour $+\infty$ et 1 pour $-\infty$.
Les nombres réservés dont $\pm\infty$ ont pris la place sont $\pm 2^{1024}$.

— **NaN :** Le sigle *NaN* est une abréviation de *Not a Number*. Dans certains langages de programmation (dont Python avec certains modules comme Numpy), NaN sert à désigner un résultat numérique impossible comme le résultat d'une division par 0 par exemple.
Les places réservées à NaN parmi les flottants sont celles dont la puissance (en codage binaire par excès) est remplie de bit 1 et la mantisse est non nulle.



Remarque 3.

Il existe d'autres places réservées : celles dont la puissance est remplie de 0 (donc $p = -1023$) et la mantisse est non nulle. Il s'agit des *nombres dénormalisés*. Si x est dénormalisé, au lieu du calcul via l'écriture scientifique binaire qui aurait dû être $\pm\overline{1,M}^2 \times 2^{-1013}$, on convient que $x = \pm\overline{\mathbf{0},M}^2 \times 2^{\mathbf{-1012}}$ ce qui permet de représenter des nombres encore plus petits.

4. Précision des calculs en flottants

Comme on l'a dit en introduction, on ne peut pas représenter l'ensemble des nombres réels par des nombres à virgule flottante, seul un ensemble fini de dyadique peut l'être. On vu également que le nombre $0,1$ n'est pas dyadique. Or si on demande à Python :

```
>>>type(0.1)
<class 'float'>
```

Pourquoi Python arrive-t-il à considérer $0,1$ comme un flottant ? Eh bien la réponse est simple : Python nous ment ! La preuve en est : testons notre fonction `dya_10_vers_bin` de l'exercice 11 sur `0.1`.

```
>>>dya_10_vers_bin(0.1)
'0,0001100110011001100110011001100110011001100110011001101'
```

Si le `0.1` de Python valait bien notre bon vieux $0,1$, nous aurions dû attendre indéfiniment la réponse car $0,1$ n'est pas dyadique !
Or si on considère la mantisse du résultat donné par `dya_10_vers_bin(0.1)`, celle-ci fait exactement 52 bits... tiens, tiens le maximum pour un flottant double précision. Donc si un nombre n'est pas dyadique, la mantisse de sa représentation en flottant est tronquée à 52 bits et arrondie à la valeur la plus proche du nombre (pour Python, qui suit une des règles possibles décrites par la norme IEEE-754) ;

À cause des arrondis et des "mensonges" de Python, on arrive à des résultats qui peuvent paraître aberrants au premier regard, mais qui sont parfaitement quantifiables quand on sait qu'on a affaire à des approximations dyadiques !

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

Conclusion : lorsqu'on travaille avec les flottants, il faut bannir les tests du type `float1 == float2` ou `float1 != float2` après des opérations sur ceux-ci. Par exemple :

Pas bien !

```
C=10000
while C != 0.0:
    C -= 0.1
print('Décollage !')
```

Biiiien !

```
C=10000
eps = 10**(-5) # valeur à estimer soit même en tenant compte des arrondis
while abs(C) <= eps:
    C -= 0.1
print('Décollage !')
```