CODER UN RÉEL FLOTTANT EN BINAIRE

I) Écrire un nombre à virgule en binaire

1) Base 2 vers base 10

En base 10, on a par exemple :

123,456 = ☐ × 10^2 + ☐ × 10^1 + ☐ × 10^0 + ☐ × 10^{-1} + ☐ × 10^{-2} + ☐ × 10^{-3}

De même, en base 2, on aura :

$10{,}1101_2$ = ☐ × 2^1 + ☐ × 2^0 + ☐ × 2^{-1} + ☐ × 2^{-2} + ☐ × 2^{-3} + ☐ × 2^{-4}

2) Base 10 vers base 2

a) En décomposant la partie décimale en somme de puissances négatives de 2

Les premières puissances négatives de 2 sont :

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}
0,5					

Ex : 0,15625 = 2^ + 2^ = $_2$

b) En effectuant des multiplications par 2

0,15625×2 = **0**,3125
0,3125×2 = **0**,625
0,625×2 = **1**,25

☐ = ☐,☐
☐ = ☐,☐

donc 0,15625 = ☐

Explication :

Refaisons exactement les mêmes calculs que ci-dessus mais directement en base 2 :

0,00101×10 = **0**,0101
0,0101×10 = **0**,101
0,101×10 = **1**,01
0,01×10 = **0**,1
0,1×10 = **1**

3) Nombres ayant une infinité de décimales en binaire

Ex : Convertissons 0,1 en binaire

0,1×2 = 0,2
0,2×2 = 0,4
0,4×2 = 0,8
0,8×2 = 1,6
0,6×2 = 1,2
0,2×2 = 0,4

La 6ème ligne est identique à la 2ème !
Le processus de conversion en base 2 ne s'arrêtera donc jamais :

0,1 = 0,00011001100110011……

Le nombre 0,1 a donc en binaire une infinité de chiffres après la virgule et devra donc être arrondi.

Conséquence :
Taper les 2 commandes ci-dessous dans la console Python :

```
>>> 0.1+0.2
```

```
>>> 0.1+0.2 == 0.3
```

On ne comparera donc jamais directement des flottants !

II) Norme IEEE754

1) Le problème du codage de la virgule :

On a vu précédemment comment coder 0,15625 en binaire :

$$0{,}15625_{10} = 0{,}00101_2$$

Mais comment peut-on coder la virgule, sachant que l'ordinateur ne connaît pas d'autres symboles que le 0 et le 1 ?

Une première solution, appelée codage en **virgule fixe**, consiste à décider que tous les nombres seront codés par exemple sur 16 bits avec les 10 premiers bits pour la partie entière et les 6 derniers pour la partie décimale. Ce système permet des calculs plutôt simples mais n'est pas adapté pour manipuler à la fois des très grands nombres et des tous petits. Il est donc peu utilisé de nos jours.

Du coup une autre solution est apparue, appelée codage en **virgule flottante**, qui consiste à « décaler » la virgule comme on le fait en notation scientifique :

$$0{,}00101_2 = 1{,}01_2 \times 2^{-3}$$

toujours 1 — mantisse — exposant à coder lui aussi en binaire

Bien sûr, le calcul en virgule flottante suppose des circuits spécialisés assez complexes, mais il permet de manipuler une plage de nombres bien plus importante que le calcul en virgule fixe, tout en gardant une meilleure précision. Du coup, la quasi totalité des processeurs travaillent aujourd'hui en virgule flottante en utilisant la norme IEEE754. Cette norme permet de coder un nombre décimal sur 32 bits (simple précision), 64 bits (double précision) ou même 128 bits (quadruple précision).

Sur la plupart des interpréteurs Python, les nombres de type float sont codés en double précision.

2) Format « simple précision » sur 32 bits

s	Exposant + 127 (8 bits)	Mantisse (23 bits)

Bit de signe :
0 : positif
1 : négatif

Le décalage de 127 permet de représenter aussi les exposants négatif

Ex 1 : Représenter −0,15625 en simple précision

−0,15625 est négatif donc le bit de signe est ☐

$0{,}15625 = 0{,}125 + 0{,}03125 = 2^{-3} + 2^{-5} = 0{,}00101_2 = 1{,}01_2 \times 2^{-3}$

La mantisse est ☐

Codons l'exposant : ☐ + 127 = ☐ = 64 + 32 + 16 + 8 + 4 = $0111\ 1100_2$

Bilan : −0,15625 s'écrit en simple précision : 1 01111100 01000000000000000000000

Ex 2 : Quel nombre s'écrit en simple précision : 0 10000001 01110000000000000000000 ?

Le bit de signe est 0 donc ce nombre est ☐

La mantisse est ☐

Décodons l'exposant : $10000001_2 = 2^0 + 2^7 = 1 + 128 = 129$

Or 129 − ☐ = ☐

Donc le nombre cherché est : ☐$_2 \times 2^{\square}$ = (1+0,25+0,125+0,0625) × 4 = 1,4375 × 4 = 5,75

Convertisseur en ligne : https://www.h-schmidt.net/FloatConverter/IEEE754.html

3) Format « double précision » sur 64 bits

s	Exposant + 1023 (11 bits)	Mantisse (52 bits)

4) Format « quadruple précision » sur 128 bits

s	Exposant + 16383 (15 bits)	Mantisse (112 bits)