Pourquoi ma calculatrice se trompe-t-elle ?

Enquête sur la représentation binaire des nombres

Activité SNT – Seconde

Mise en situation. En console Python de la calculatrice, on tape le calcul suivant :

$$0,1 \times 3 \times 10^{16}$$

On s'attend bien sûr à obtenir 3 000 000 000 000 000 (soit 3×10^{15}). Or la calculatrice affiche :

```
>>> 0.1*3*10^16
3000000000000001
```

Comment expliquer ce résultat surprenant, alors que la calculatrice ne "sait" pas se tromper dans un calcul aussi simple ? C'est ce que nous allons comprendre pas à pas dans cette activité, en remontant jusqu'au langage même que parlent les ordinateurs et calculatrices : le **binaire**.

Objectifs de l'activité :

- Comprendre le système de numération binaire et savoir convertir entre base 10 et base 2.
- Comprendre comment un ordinateur stocke un nombre décimal en mémoire (nombres à virgule flottante).
- Découvrir que certains nombres décimaux « simples » n'ont pas d'écriture binaire finie.
- Expliquer, à l'aide de ces notions, le résultat obtenu sur la console python.

1 Le binaire : le langage des machines

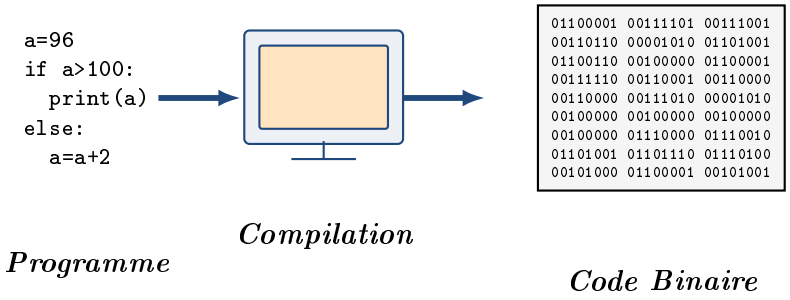
Comment parler le même langage que les ordinateurs ?

La mémoire des ordinateurs (et donc des calculatrices) est constituée d'une multitude de petits circuits électroniques qui, chacun, ne peuvent être que dans **deux états** : allumé ou éteint. Ces deux états sont représentés par les valeurs **0** ou **1**.

Définition

Une telle valeur, 0 ou 1, s'appelle un **booléen**, un **chiffre binaire** (nombre en base 2), ou encore un **bit** (*binary digit*). Un groupe de 8 bits est appelé un **octet**, et deux octets forment un **mot**.

Lorsqu'un programmeur écrit un programme (par exemple en Python), celui-ci est ensuite traduit par l'ordinateur, lors de la **compilation**, en une longue suite de 0 et de 1 :



Programme *Compilation* *Code Binaire*

Les programmes python mais aussi les textes, les nombres, les sons, les images s'expriment comme des suites de bits (0 ou 1).

Les textes, les nombres, les sons et les images s'expriment donc tous, au sein d'une machine, sous la forme de longues suites de bits. Pour comprendre ce que fait réellement une calculatrice quand elle effectue un calcul, il est indispensable de savoir comment elle représente les nombres dans ce langage binaire, et comment on convertit un nombre binaire en nombre décimal (et réciproquement).

Définition

Nous comptons habituellement en **base 10** (système décimal) : nous utilisons 10 chiffres (0 à 9). Dans le système binaire, on n'utilise que deux chiffres, 0 et 1, mais le principe de position reste le même : chaque position d'un chiffre correspond à une puissance de la base.

A. Conversion d'un nombre binaire en base 10

Dans le système décimal, chaque position d'un chiffre correspond à une puissance de 10. Par exemple :

$$2503 = 2 \times 10^3 + 5 \times 10^2 + 0 \times 10^1 + 3 \times 10^0$$

De la même façon, dans le système binaire, chaque position correspond à une puissance de 2. Pour convertir un octet (8 bits) de la base 2 vers la base 10, on écrit l'octet dans un tableau où chaque colonne représente une puissance de 2 :

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

On place ensuite chaque bit de l'octet sous la puissance de 2 correspondante, puis on additionne les puissances de 2 pour lesquelles le bit vaut 1.

Exemple : convertir l'octet 00101011 en base 10

$2^7 = 128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
0	0	1	0	1	0	1	1

On additionne uniquement les puissances de 2 correspondant aux bits égaux à 1 :

$$00101011_2 = 32 + 8 + 2 + 1 = 43$$

▶ À vous de jouer !

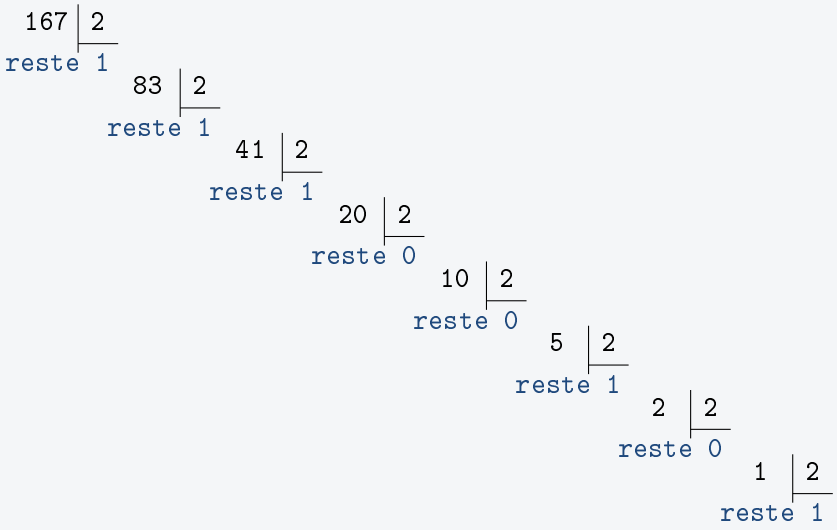
1. En utilisant le même type de tableau, convertir en base 10 le nombre binaire 1101_2.
2. Convertir en base 10 le nombre binaire 100000_2. Que remarque-t-on ?

B. Conversion d'un nombre décimal en base 2

Pour convertir un nombre décimal en binaire, on effectue des **divisions euclidiennes successives par 2** : le quotient obtenu à chaque étape devient le dividende de l'étape suivante, et on s'arrête lorsque le quotient est nul. On remonte ensuite la suite des restes obtenus, **du dernier au premier**, pour lire l'écriture binaire.

Exemple détaillé : convertir 167 en base 2

On pose les divisions euclidiennes successives par 2 les unes en dessous des autres, en escalier :



Ce qui nous donne :

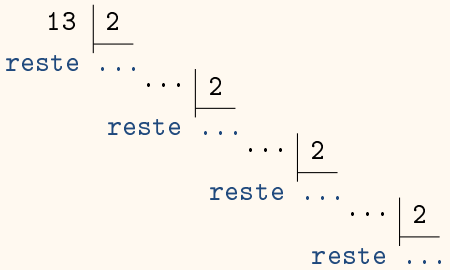
$$\begin{aligned} 167 &= 2 \times 83 + 1 \\ 83 &= 2 \times 41 + 1 \\ 41 &= 2 \times 20 + 1 \\ 20 &= 2 \times 10 + 0 \\ 10 &= 2 \times 5 + 0 \\ 5 &= 2 \times 2 + 1 \\ 2 &= 2 \times 1 + 0 \\ 1 &= 2 \times 0 + 1 \end{aligned}$$

On lit maintenant la colonne des restes **du dernier au premier** (de bas en haut) :

$$167 = 10100111_2$$

▶ À vous de jouer !

En reproduisant très précisément la même méthode (divisions euclidiennes successives par 2 posées en escalier, puis lecture des restes de bas en haut), convertir 13 en binaire. Complétez le schéma suivant :



Coup de pouce

On pose : $13 = 2 \times 6 + 1$, puis $6 = 2 \times 3 + 0$, puis $3 = 2 \times 1 + 1$, puis $1 = 2 \times 0 + 1$. On lit les restes obtenus en partant du dernier vers le premier.

Remarque

Voici, à titre de référence, les écritures binaires des entiers de 0 à 14 :

10	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	0	1	10	11	100	101	110	111	1000	1001	1010	1011	1100	1101	1110

2 Écrire des nombres non entiers en binaire

Tout comme en base 10 on peut utiliser des chiffres après la virgule associés à des puissances négatives de 10 ($0,25 = 2\times10^{-1}+5\times10^{-2}$), on peut écrire des nombres non entiers en binaire à l'aide de puissances négatives de 2.

Définition

En binaire, les chiffres après la virgule sont associés aux puissances négatives de 2 :

$$2^{-1} = 0,5 \quad ; \quad 2^{-2} = 0,25 \quad ; \quad 2^{-3} = 0,125 \quad ; \quad 2^{-4} = 0,0625 \ldots$$

Exemple

Le nombre binaire $0,101_2$ signifie :

$$1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3} = 0,5 + 0 + 0,125 = 0,625$$

La méthode des multiplications successives par 2

Pour convertir un nombre décimal compris entre 0 et 1 en binaire, on procède ainsi :

1. On multiplie le nombre par 2.
2. Le résultat obtenu a une partie entière (0 ou 1) : c'est le **premier chiffre binaire** après la virgule.
3. On isole la nouvelle partie décimale (ce qui reste après la virgule) et on recommence l'opération sur celle-ci.
4. On répète le procédé autant de fois que nécessaire pour obtenir le nombre de chiffres souhaité.

Exemple détaillé : convertir $0,625$ en binaire

On applique la méthode pas à pas. À chaque étape, on multiplie par 2 la partie décimale de l'étape précédente, on note le chiffre entier obtenu (0 ou 1), puis on ne garde que la partie décimale pour l'étape suivante.

Étape	Calcul	Chiffre obtenu	Partie décimale restante
1	$0,625\times2 = 1,25$	**1**	$0,25$
2	$0,25\times2 = 0,5$	**0**	$0,5$
3	$0,5\times2 = 1,0$	**1**	0

La partie décimale restante est nulle : le processus s'arrête. On lit les chiffres obtenus **de haut en bas** (dans l'ordre où ils ont été trouvés) :

$$0,625 = 0,101_2$$

On retrouve bien le résultat de l'exemple précédent !

Remarque

Attention à ne pas confondre avec la méthode des divisions par 2 utilisée pour les nombres entiers (partie 1), où l'on lisait les restes **de bas en haut**. Ici, pour la partie décimale, on lit les chiffres obtenus **de haut en bas**, dans l'ordre où on les calcule.

À votre tour : convertir 0,1 en binaire

Nous allons maintenant appliquer très précisément cette méthode au nombre qui nous intéresse : $0,1$.

▶ À vous de jouer !

On souhaite convertir $0,1$ (en base 10) en binaire, en suivant scrupuleusement la méthode ci-dessus. Complétez le tableau suivant en détaillant **tous** les calculs (comme dans l'exemple précédent), sur au moins 8 étapes.

Étape	Calcul	Chiffre obtenu	Partie décimale restante
1	$0,1 \times 2 = \ldots$	$\ldots$	$\ldots$
2	$\ldots \times 2 = \ldots$	$\ldots$	$\ldots$
3			
4			
5			
6			
7			
8			

1. Complétez entièrement le tableau ci-dessus, étape par étape, sans sauter aucun calcul.
2. La partie décimale restante s'annule-t-elle à un moment ? Que peut-on en déduire sur le nombre d'étapes nécessaires pour écrire $0,1$ en binaire ?
3. En observant la colonne « chiffre obtenu », repérez un motif (une suite de chiffres) qui se répète. Écrivez ce motif.
4. En déduire l'écriture binaire de $0,1$ sous la forme $0,\underbrace{\ldots\ldots\ldots\ldots}_{\text{motif répété}}\ldots{}_2$

Coup de pouce

Voici les deux premières étapes entièrement détaillées pour vous aider à démarrer :

- **Étape 1 :** $0,1 \times 2 = 0,2$. La partie entière est 0, la partie décimale restante est $0,2$.
- **Étape 2 :** $0,2 \times 2 = 0,4$. La partie entière est 0, la partie décimale restante est $0,4$.

Continuez ainsi de suite : à l'étape 3, il faudra multiplier $0,4$ par 2, etc. Soyez attentif : dès que la partie décimale restante dépasse $0,5$, la multiplication par 2 donnera un résultat supérieur à 1, et le chiffre obtenu sera alors un **1**.

Remarque

Vous devriez observer que la partie décimale restante **ne s'annule jamais**, et que le motif **0011** se répète indéfiniment à partir d'un certain rang. On obtient finalement :

$$0,1 \text{ (en base 10)} \;=\; 0,0001100110011001100110011\ldots_2$$

Le nombre $0,1$ n'a donc pas d'écriture binaire finie : c'est un nombre binaire "infini périodique" ! Ce phénomène est comparable au fait que $\frac{1}{3} = 0,333\ldots$ n'a pas d'écriture décimale finie : certains nombres qui paraissent "simples" dans une base ne le sont plus du tout dans une autre base.

3 Comment la machine stocke les nombres

Une calculatrice ou un ordinateur ne dispose pas d'une mémoire infinie : chaque nombre est stocké sur un **nombre fixe de bits** (souvent 64 bits pour les nombres décimaux, appelés « nombres à virgule flottante »). Ces 64 bits se répartissent ainsi :

1 bit	**11 bits**	**52 bits**
signe (+ ou −)	exposant (puissance de 2)	mantisse (chiffres binaires significatifs)

Définition

On appelle cette écriture la **notation scientifique binaire** : tout nombre est mis sous la forme

$$\pm 1,\underbrace{b_1 b_2 b_3 \ldots b_{52}}_{\text{mantisse}} \times 2^e$$

Puisque seuls **52 bits** sont réservés à la mantisse, **tout chiffre binaire au-delà du 52e doit être supprimé (tronqué) ou arrondi** : la machine ne peut pas stocker un nombre binaire infini !

Remarque

C'est exactement comme si, avec une calculatrice qui n'affiche que 10 chiffres, vous deviez arrondir $\frac{1}{3} = 0,3333333333\ldots$ à $0,3333333333$: on perd un peu de précision, mais l'erreur est en général invisible... sauf si elle est amplifiée par la suite !

4 Résolution de l'énigme

Nous avons maintenant tous les outils pour comprendre le résultat de la console python.

▶ À vous de jouer !

1. D'après la partie précédente, le nombre $0,1$ ne peut pas être stocké **exactement** en mémoire, puisque son écriture binaire est infinie. La calculatrice stocke donc une valeur légèrement différente de $0,1$, que l'on note $0,1^*$. À votre avis, $0,1^*$ est-il légèrement plus grand ou légèrement plus petit que $0,1$?
2. Lorsqu'on calcule $0,1^* \times 3$, l'écart (l'erreur) entre le résultat obtenu et la vraie valeur $0,3$ est-il plus grand, plus petit, ou égal à l'écart entre $0,1^*$ et $0,1$? Justifier intuitivement.
3. Enfin, lorsqu'on multiplie le résultat précédent par 10^{16} (un très grand nombre), que devient un tout petit écart initialement invisible ?

Coup de pouce

Pour la question 3 : si l'écart initial est de l'ordre de 10^{-17} (extrêmement petit devant $0,1$), en le multipliant par 10^{16}, on obtient un écart de l'ordre de... $10^{-17} \times 10^{16} = 10^{-1} = 0,1$! Un écart invisible peut donc devenir visible si on multiplie par un nombre suffisamment grand : c'est le principe de **l'amplification des erreurs d'arrondi**.

Bilan mathématique

En réalité, la valeur $0,1^*$ réellement stockée en mémoire (sur 52 bits de mantisse) vaut très précisément :

$$0,1^* = 0,1000000000000000055511151231257827\ldots$$

soit un écart avec $0,1$ d'environ $5,55 \times 10^{-18}$. Après multiplication par 3, puis par 10^{16}, cet écart minuscule se retrouve multiplié par 3×10^{16}, ce qui donne un écart final de $0,5$: le résultat calculé par la machine est en réalité

$$0,1^* \times 3 \times 10^{16} = 3\,000\,000\,000\,000\,000,5$$

Or la console python n'affiche que 16 chiffres significatifs : la valeur $3\,000\,000\,000\,000\,000,5$ doit donc être **arrondie**. Comme elle se situe exactement entre $3\,000\,000\,000\,000\,000$ et $3\,000\,000\,000\,000\,001$, l'arrondi (à la règle "au plus proche, ou au supérieur en cas d'égalité") retenu par la calculatrice donne :

$$\boxed{3\,000\,000\,000\,000\,001}$$

Voilà l'origine du fameux « 1 » final !

▶ À vous de jouer !

Pour conclure cette enquête, répondre par une phrase à chacune des questions suivantes :

1. Pourquoi une calculatrice ne peut-elle pas toujours représenter exactement un nombre décimal ?
2. Pourquoi cette imprécision est-elle presque toujours invisible dans la vie de tous les jours ?
3. Dans quel type de calcul risque-t-on de "voir" apparaître ce genre d'erreur ?

Pour aller plus loin

Ce phénomène s'appelle en informatique une **erreur d'arrondi en virgule flottante** (*floating-point rounding error*). Il est décrit par la norme internationale **IEEE 754**, utilisée par la quasi-totalité des calculatrices et ordinateurs du monde. Ce n'est donc pas un bug de la calculatrice : c'est une conséquence universelle et incontournable du codage binaire des nombres réels !