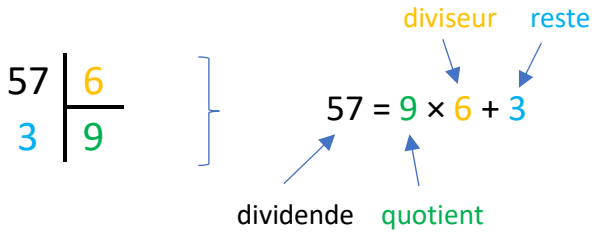
Chapitre 5 : Représentation des nombres

1. Représentation d'un nombre

La **division euclidienne** (ou **division entière**) est une procédure de calcul qui, à deux entiers naturels appelés dividende et diviseur, associe deux autres entiers appelés quotient et reste :



1.1. Écriture d'un entier positif [Voir chapitre 1]

Le système de numération que nous utilisons naturellement pour compter ou représenter les nombres est le système décimal utilisant la base 10 qui s'appuie sur l'utilisation de **10 symboles** (base 10) : 0, 1, 2, 3, 4, 5, 6, 7, 8 et 9.

Exemple : $253_{décimal} = 2 \times 10^2 + 5 \times 10^1 + 3 \times 10^0$

En informatique, les bases les plus utilisées sont les bases 2 (binaire) et 16 (hexadécimale).

- **La base binaire (base 2) :**

Ce système de numération s'appuie sur l'utilisation de **2 symboles** (base 2), appelés « **bit** » (de l'anglais *binary digit*, soit « chiffre binaire »), qui peuvent prendre deux valeurs, notées par convention **0** et **1**.

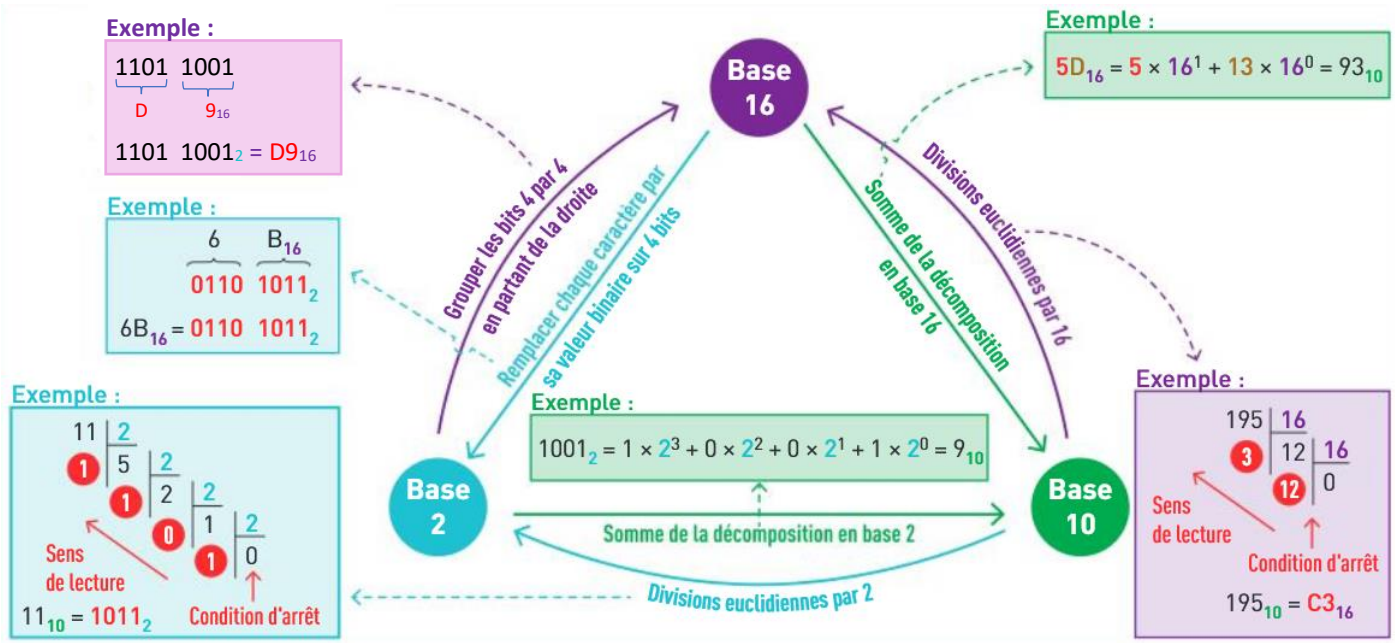
Exemple : $253_{décimal} = 1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 11111101_{binaire}$

- **La base hexadécimale (base 16) :**

Ce système de numération s'appuie sur l'utilisation de **16 symboles** (base 16), appelés chiffres hexadécimaux : **0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, **8**, **9**, **A**, **B**, **C**, **D**, **E** et **F**.

Exemple : $253_{décimal} = 15 \times 16^1 + 13 \times 16^0 = FD_{hexadécimal}$

1.2. Conversions d'un entier positif « à la main » :



1.3. Conversion d'un entier positif avec Python

Des fonctions natives en Python permettent la conversion des nombres entiers entre les différentes bases (2, 10 et 16) :

```
>>> bin(56)    #conversion d'un entier en binaire
'0b111000'     #le préfixe « 0b » indique que le nombre est exprimé en base binaire
>>> hex(56)    #conversion d'un entier en hexadécimal
'0x38'         #le préfixe « 0x » indique que le nombre est exprimé en base hexadécimale
```

```
>>> int('0b111000', 3)    #conversion d'un nombre binaire (base 2) en décimal
56
>>> int('0x38', 16)       #conversion d'un nombre hexadécimal (base 16) en décimal
56
```

2. Nombre de bits pour l'écriture d'un entier en binaire

2.1. Somme de deux entiers positifs

Avant de représenter (écrire) un entier, il est nécessaire de définir le nombre de bits qui seront utilisés pour cette représentation (souvent 8, 16, 32 ou 64 bits).

Avec n bits, on peut représenter 2^n valeurs possibles : le plus grand entier positif que l'on peut représenter est $2^n - 1$ (car le premier entier est 0)

Pour déterminer le nombre de bits minimum nécessaires pour écrire (ou stocker ou représenter) un entier positif en base 2, il faut déterminer la plus petite puissance de 2 qui permet d'écrire un nombre d'entiers positif strictement supérieur à l'entier à écrire.

Exemple : combien de bits sont nécessaires pour représenter 210 en base 2 ?

$$2^7 = 128_{10} \quad ; \quad 2^8 = 256_{10} \quad ; \quad 128 < 210 < 256$$

$\Rightarrow$ Pour représenter 210_{10} en base 2, il faut utiliser 8 bits : $2^7 < 210_{10} < 2^8$

Pour représenter la somme de deux entiers positifs, représentés sur n bits, sans connaitre le résultat préalable, il faut utiliser n + 1 bits.

Démonstration :

❶ Sur n bits, le plus grand entier positif que l'on peut représenter a pour valeur : $2^n - 1$

❷ La plus petite somme possible de deux entiers représentés sur n bits est $(2^n - 1) + (2^n - 1) = 2 \times (2^n - 1)$:

$$2 \times (2^n - 1) = 2 \times 2^n - 2 = 2^{n+1} - 2 \quad \text{et} \quad 2^n < 2^{n+1} - 2 < 2^{n+1}$$

Exemple :

❶ 255 est le plus grand entier positif que l'on peut représenter avec 8 bits ($2^8 - 1 = 255$). 511 est le plus grand entier positif que l'on peut représenter avec 9 bits ($2^9 - 1 = 511$) ;

❷ 255 + 255 = 510 donc pour représenter la somme de deux entiers représentés sur 8 bits il faut utiliser 9 bits car :

$$2^8 < 510 < 2^9$$

$\Rightarrow$ La somme de deux entiers positifs représentés sur 8 bits est représentable sur 9 bits (= 8 bits + 1 bit).

2.2. Produit de deux entiers positifs

Pour représenter le produit de deux entiers positifs, représentés sur n bits, sans connaitre le résultat préalable, il faut utiliser 2n bits.

Démonstration :

❶ Sur n bits, le plus grand entier positif que l'on peut représenter a pour valeur : $2^n - 1$

❷ Le plus grand produit, de deux entiers représentés sur n bits, que l'on peut représenter est $(2^n - 1) \times (2^n - 1)$:

$$(2^n - 1) \times (2^n - 1) = (2^n - 1)^2 = (2^n)^2 - 2 \times 2^n \times 1 + 1^2 = 2^{2n} - 2^{n+1} + 1 \text{ et } \quad 2^n < 2^{2n} - 2^{n+1} + 1 < 2^{2n}$$

Rappel de mathématique (identités remarquables) : $(a - b)^2 = a^2 - 2ab + b^2$

Exemple :

❶ 255 est le plus grand entier positif que l'on peut représenter avec 8 bits ($2^8 - 1 = 255$) ;
32768 est le plus grand entier positif que l'on peut représenter avec 15 bits ($2^{15} - 1 = 32767$) ;
65535 est le plus grand entier positif que l'on peut représenter avec 16 bits ($2^{16} - 1 = 65535$) ;

❷ $255 \times 255 = 65025$ donc pour représenter le produit de deux entiers représentés sur 8 bits il faut utiliser $2 \times 8 = 16$ bits car :

$$2^{15} < 65025 < 2^{16}$$

⇒ La produit de deux entiers positifs représentés sur 8 bits est représentable sur 16 bits (= 8 bits + 8 bits).

3. Addition d'entiers positifs en binaire

Pour additionner des nombres binaires, il suffit d'additionner les nombres en base 2 en suivant le même mode opératoire qu'en base 10 et en utilisant la table d'addition suivante :

$0_2 + 0_2 = 0_2$	Table d'addition en binaire
$0_2 + 1_2 = 1_2$	
$1_2 + 0_2 = 1_2$	
$1_2 + 1_2 = {}^1 0$	(on pose 0 et on retient 1)
$1_2 + 1_2 + 1_2 = {}^1 1$	(on pose 1 et on retient 1 ⇔ ${}^1 0 + 1_2 = {}^1 1$)

Exemple :

retenues → 11 11

	10110011	179
+	00011001	25
	11001100	204

On a donc en binaire :

$10110011_2 + 00011001_2 = 11001100_2$

Pour plus d'informations :

- https://fr.wikihow.com/additionner-des-nombres-binaires.

4. Représentation d'entiers relatifs

Pour représenter un entier relatif signé (positifs ou négatifs) en binaire, il faut utiliser la représentation en **complément à 2^n** (ou « **complément à 2 sur n bits** » ou « **complément à 2** »).

Méthode : complément à 2^n

❶ Il est nécessaire de définir tout d'abord le nombre n de bits qui seront utilisés pour cette représentation (8, 16, 32 ou 64 bits, en général) ;

❷ On représente le nombre positif en binaire, en le précédent d'autant de 0 qu'il faut pour avoir n bits au total ;

❸ On fait le « complément à 1 » : on inverse tous les bits (les bits à 1 passent à 0 et vice versa).

❹ On additionne 1 (en binaire, sur n bits) au nombre binaire obtenu à l'étape précédente (en respectant les règles d'addition des nombres binaires) ;

Exemple : qu'elle est la représentation de -24_{10} sur 8 bits ?

0001 1000	←	24_{10} en binaire sur 8 bits
11	←	Retenues
1110 0111	←	Complément à 1
0000 0001	←	Addition de 1 en binaire
1110 1000	←	-24_{10} en binaire sur 8 bits

La représentation de -24_{10} sur 8 bits est donc

1110 1000

Remarque : $24_{10} = 11000_2$

Vérifions que $24_{10} + (-24_{10}) = 0_{10}$ en binaire :

1111	←	Retenues
0001 1000	←	24^{10} en binaire sur 8 bits
1110 1000	←	-24_{10} en binaire sur 8 bits
0000 0000	←	0_{10} en binaire sur 8 bits

→ Dans l'addition ci-dessus, nous avons un 1 pour le 9ème bit, mais comme la représentation se limite à 8 bits, il nous reste bien 00000000.

Méthode rapide :

Pour représenter un nombre en son complément à 2^n, sans poser de calcul, il suffit de garder tous les bits depuis la droite jusqu'au premier « 1 » compris puis d'inverser tous les bits qui sont à gauche de ce premier 1.

Complément à 2^n

0001 1000 → 1110 1000

Remarques :

- Pour déterminer si une représentation correspond à un entier relatif positif ou un entier relatif négatif, il suffit de regarder le bit de poids fort (le bit le plus à gauche) : s'il est à 1 alors l'entier est négatif, s'il est à 0 alors l'entier est positif ;
- La plus petite valeur qu'il est possible de représenter sur 8 bits est 1000 0000 (soit –128) et la plus grande valeur est 0111 1111 (soit +127).

5. Représentation des nombre réels

Compte tenu du nombre limité de bits en mémoire, tous les nombres réels ne sont pas exactement représentables en binaire car certains ont une représentation binaire infinie.

Méthode pour convertir un nombre décimal en binaire :

❶ On représente d'abord la partie entière en binaire ;

❷ On multiplie par 2 la partie décimale : la partie entière du nombre obtenu donne le prochain chiffre binaire après la virgule du nombre convertit en binaire ;

❸ On ne garde que la partie décimale du nombre obtenu à l'étape ❷

❹ On recommence les étapes ❷ et ❸ avec cette nouvelle partie décimale, jusqu'à obtenir une partie décimale nulle.

Exemple : comment représenter « $5{,}1875_{10}$ » en binaire ?

- On commence par représenter la partie entière en binaire : $5_{10} = 101_2$
- On multiplie 0,1875 par 2 : $0{,}1875 \times 2 = 0{,}375 = 0 + 0{,}375$
- On multiplie 0,375 par 2 : $0{,}375 \times 2 = 0{,}750 = 0 + 0{,}750$
- On multiplie 0,750 par 2 : $0{,}75 \times 2 = 1{,}5 = 1 + 0{,}5$

→ Quand le résultat de la multiplication par 2 est supérieur à 1, on garde uniquement la partie décimale.

- On multiplie 0,5 par 2 : $0{,}5 \times 2 = 1{,}0 = 1 + 0{,}0$

→ La partie décimale est à **0** (0,**0**), on arrête le processus.

⇒ On obtient une succession de « a + 0,b » : 0 + 0,375, 0 + 0,75, 1 + 0,5 et 1 + 0,0.

Il suffit maintenant de retenir tous les « a » (dans l'ordre de leur obtention) afin d'obtenir la partie décimale de notre nombre : 0011

$$(5{,}1875)_{10} = (101{,}0011)_2$$

Remarque : **les nombres réels n'étant pas rigoureusement représentés en binaire, à cause des limitations du nombre de bits, il faut éviter de tester l'égalité entre deux nombres flottants (= nombres réels en programmation).**

```
1  >>> 0.1 + 0.2 == 0.3
2  False
```

Programme :

Représentation des données : types et valeurs de base

Contenus	Capacités attendues	Commentaires
Écriture d'un entier positif dans une base $b \geqslant 2$	Passer de la représentation d'une base dans une autre.	Les bases 2, 10 et 16 sont privilégiées.
Représentation binaire d'un entier relatif	Évaluer le nombre de bits nécessaires à l'écriture en base 2 d'un entier, de la somme ou du produit de deux nombres entiers. Utiliser le complément à 2.	Il s'agit de décrire les tailles courantes des entiers (8, 16, 32 ou 64 bits). Il est possible d'évoquer la représentation des entiers de taille arbitraire de Python.
Représentation approximative des nombres réels : notion de nombre flottant	Calculer sur quelques exemples la représentation de nombres réels : 0.1, 0.25 ou 1/3.	0.2 + 0.1 n'est pas égal à 0.3. Il faut éviter de tester l'égalité de deux flottants. Aucune connaissance précise de la norme IEEE-754 n'est exigible.
Valeurs booléennes : 0, 1. Opérateurs booléens : and, or, not. Expressions booléennes	Dresser la table d'une expression booléenne.	Le ou exclusif (xor) est évoqué. Quelques applications directes comme l'addition binaire sont présentées. L'attention des élèves est attirée sur le caractère séquentiel de certains opérateurs booléens.
Représentation d'un texte en machine. Exemples des encodages ASCII, ISO-8859-1, Unicode	Identifier l'intérêt des différents systèmes d'encodage. Convertir un fichier texte dans différents formats d'encodage.	Aucune connaissance précise des normes d'encodage n'est exigible.

Sitographie :

- https://pixees.fr/informatiquelycee/prem/c6c.html
- https://pixees.fr/informatiquelycee/prem/c7c.html