- Codage des entiers relatifs et des nombres réels -

1) Représentation d'un nombre dans une base $b \geqslant 2$:

La base 10 :

C'est la base dans laquelle nous représentons les nombres habituellement.
Elle utilise les **10 symboles** 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 et 9 ,
et **les puissances de 10**.

L'écriture d'un entier en base 10 correspond au schéma suivant :

$$5\,397_{10} = 5\times10^3 + 3\times10^2 + 9\times10^1 + 7\times10^0 = 5\times1\,000 + 3\times100 + 90 + 7$$

- Combien de fois peut-on diviser le nombre précédent par 10 ?
- Combien de chiffres sont-ils nécessaires dans cette écriture décimale ?
- Combien de fois peut-on diviser par 10 un décimal qui s'écrit avec 8 chiffres ?
- Comment reconnaît-on un nombre divisible par 10 en écriture décimale ?

La base 5 :

Elle utilise les **5 chiffres** 0 , 1 , 2 , 3 et 4 ,
et les **puissances de 5**.

$$23_5 = 2\times5^1 + 3\times5^0 = 2\times5 + 3\times1 = 13_{10}$$
$$681_5 = 6\times5^2 + 8\times5^1 + 1\times5^0 = 6\times25 + 8\times5 + 1 = 191_{10}$$

Ecriture des premiers nombres décimaux en base 5																				
Base 5	*0*	*1*	*2*	*3*	*4*	*10*	*11*	*12*	*13*	*14*	*20*	*21*	*22*	*23*	*24*	*30*	*31*	*32*	*33*	*34*
Base 10	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

- Donner l'écriture décimale du nombre 2021_5 .
- Comment reconnaître un décimal divisible par 5 , lorsqu'il est écrit en base 5 ?
- Combien de fois peut-on diviser par 5 le nombre 1789_5 ?
- Quel est le plus grand décimal qu'on puisse écrire en base 5 avec 8 chiffres ?

Conversion base décimale ==> base 5 :

Méthode 1 : Pour 353_{10} .

On regarde les puissances successives du nombre 5 : $5^0=1$, $5^1=5$, $5^2=25$, $5^3=125$...
On essaye ensuite de décomposer le nombre en commençant par les plus grandes puissances de 5 possibles, en les prenant au maximum 4 fois :

$$353_{10} = 2\times125 + 4\times25 + 0\times5 + 3\times1 \text{ qui s'écrit } 2403_5 \text{ en base } 5 .$$

Exercice : Ecrire $3\,021_{10}$ en base 5 de cette manière.

Méthode 2 : Pour 729_{10} .

On effectue la division euclidienne de 729 par 5 ,
puis on divise le quotient obtenu par 5 ,
et on recommence avec les quotients successifs, jusqu'à obtenir un quotient égal à 0 :

729	5
22	145
29	
4	

145	5
45	29
0	

29	5
4	5

5	5
0	1

1	5
1	0

Pour écrire 729_{10} en base 5 , on écrit tous les restes obtenus en commençant par le dernier : $729_{10}=10404_5$

On vérifie bien que $10404_5 = 1\times5^4 + 0\times5^3 + 4\times5^2 + 0\times5^1 + 4\times1 = 729_{10}$.

- Combien de fois peut-on diviser le nombre précédent par 5 ?
- Combien de chiffres sont nécessaires pour son écriture en base 5 ?
- Combien de fois peut-on diviser par 5 le nombre 4444444_5 ?

Exercice : En utilisant les divisions euclidiennes, donner la représentation en base 5 des décimaux 847_{10} et 2020_{10} .

La base octale, ou base 8 :

Exercice :

- Donner l'écriture décimale des nombres 307_8 et 1524_8.
- Donner la représentation en base octale du décimal 2048_{10}.
- Comment reconnaître un décimal divisible par 8 lorsqu'il est écrit en base octale ?

La base 2 :

Elle utilise les **deux chiffres 0 et 1** ainsi que les **puissances de 2**.
La représentation d'un nombre en base 2 est appelée **représentation binaire**.

$$10\,1100_2 = 0\times2^0+0\times2^1+1\times2^2+1\times2^3+0\times2^4+1\times2^5 = 0\times1+0\times2+1\times4+1\times8+0\times16+1\times32 = 44_{10}$$
$$1001\,0111_2 = 1\times1+1\times2+1\times4+0\times8+1\times16+0\times32+0\times64+1\times128 = 1+2+4+16+128 = 151_{10}$$

Il faut bien connaître les puissances de 2 pour utiliser le binaire :

2^0	2^1	2^2	2^3	2^4	2^5	2^6	2^7	2^8	2^9	2^{10}
1	2	4	8	16	32	64	128	256	512	1024

- Dans la représentation binaire d'un nombre, le **bit de poids faible** (**LSB** en anglais pour least signifiant bit) est celui qui est le plus à droite.
- Le **bit de poids fort** (**MSB** en anglais pour most signifiant bit) est celui qui est le plus à gauche.

Dans une représentation binaire sur **1 octet** ou **8 bits** on aura :

Poids	128	64	32	16	8	4	2	1
Bits	MSB	x	x	x	x	x	x	LSB

Par exemple le binaire $1100\,1010_2$ a son MSB à 1, et son LSB à 0.

Conversion base décimale ==> base 2 :

Exemple : Conversion de 25_{10} en binaire :

25	2
1	12

12	2
0	6

6	2
0	3

3	2
1	1

1	2
1	0

Ce qui se traduit mathématiquement ainsi :

$$\begin{aligned} 25 &= 2\times 12+1 \\ &= 2\times(2\times 6+0)+1 \\ &= 2\times(2\times(2\times 3+0)+0)+1 \\ &= 2\times(2\times(2\times(2\times 1+1)+0)+0)+1 \\ &= 2\times(2\times(2\times(2\times(2\times 0+1)+1)+0)+0)+1 \end{aligned}$$

Et qui donne alors : $25_{10}=11001_2$

Exercice : En effectuant les divisions euclidiennes par 2 , donner la représentation binaire des décimaux 329_{10} et 2020_{10} , comme dans l'exemple pour la base 5 .

Questions :

- Comment reconnaît-on un nombre pair écrit en binaire ?
- Quel est l'effet de la multiplication par 2 sur la repésentation binaire d'un décimal ?
- Et pour la division par 2 si le nombre est pair ?
- Quels sont les décimaux dont l'écriture binaire est de la forme 100...00 ?
- Quel est le plus grand entier représentable sur 2 octet ou 16 bits ?
- Combien de fois peut-on diviser ce nombre par 2 ?

Algorithmes : conversion base 2 vers base 10, et base 10 vers base 2

Python : manipulation des fonctions python associées au codage binaire

La base 16 ou base hexadécimale :

Il faut **16 symboles** pour utiliser cette base.
Comme on ne dispose que de 10 chiffres, il faudra utiliser en plus de ces 10 chiffres les lettres A , B , C , D , E , F avec :

Base 10	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Base 16	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

On a alors $10_{16}=16_{10}$, $11_{16}=17_{10}$ et aussi $1A_{16}=26_{10}$ et $1F_{16}=31_{10}$.

Exemples : $5CE_{16}=5\times16^2+12\times16+14\times16^0=1280+192+14=1486_{10}$

$2B3C_{16}=2\times16^3+11\times16^2+3\times16+12\times16^0=8192+2816+48+12=11068_{10}$

Exercice : Donner l'écriture décimale des nombres $7D_{10}$ et $1A3B_{16}$.
Donner l'écriture binaire de D_{16} et de 7_{16} , puis de $7D_{16}$.
Donner l'écriture binaire de $1A3B_{16}$.

Remarque : La base hexadécimale est utile en informatique, car elle permet d'écrire les nombres binaires de manière plus condensée.

Conversion binaire ==> hexadécimal :

Comme $2^4=16$, pour représenter un nombre binaire sous forme hexadécimale, il suffit de prendre ses chiffres par groupe de 4.

Sa taille sous forme hexadécimale sera donc 4 fois plus petite que sa taille en représentation binaire.

$1111_2=(2^4-1)_{10}=15_{10}=F_{16}$

$1000\ 1010_2=8_{10}+10_{10}=8A_{16}$

$1110\ 0110\ 1101\ 0111_2=14_{10}+6_{10}+13_{10}+7_{10}=E6D7_{16}$

Décimal	Binaire 8	4	2	1	Hexadécimal
0	0	0	0	0	0
1	0	0	0	1	1
2	0	0	1	0	2
3	0	0	1	1	3
4	0	1	0	0	4
5	0	1	0	1	5
6	0	1	1	0	6
7	0	1	1	1	7
8	1	0	0	0	8
9	1	0	0	1	9
10	1	0	1	0	A
11	1	0	1	1	B
12	1	1	0	0	C
13	1	1	0	1	D
14	1	1	1	0	E
15	1	1	1	1	F

Notations : En **informatique**, on utilise les bases **décimale**, **binaire** et **hexadécimale**.
Pour différencier ces différentes représentations, on écrit:

2048	pour les représentations décimales
0b10110010	pour les représentations binaires
0x3E9A	pour les représentations hexadécimales

Exercice :

1) Donner la représentation hexadécimale des nombres suivants :

   1 000 et 35017

2) Donner la représentation héxadécimale des nombres suivants :

   0b1010110 et **0b**1011111011010110

3) Donner la représentation binaire des nombres suivants :

   0x7CB et **0x**E9AD

Algorithme : conversion base 10 vers base hexadécimale

2) Le bit informatique :

Un ordinateur tire son énergie de l'électricité.
La seule information qu"il puisse recevoir est donc de nature électrique.
Cette information est la même dans tous ses composants :

- soit un courant circule dans le circuit électronique
- soit aucun courant ne le parcourt

Cette information est donc de nature binaire : oui/non ou vrai/faux.

Au regard d'un microprocesseur, cette information est convertie en 0 ou 1.

Définition : Le bit.

Le **bit** est l'**unité d'information** compréhensible par une machine, c'est-à-dire un nombre qui vaut soit 0 soit 1. Il est la plus petite information disponible.

Remarques : Bit est la contraction de **Bi**nary Digi**t**.

Avec 1 bit on peut donc représenter 2 nombres : le 0 et le 1.
Avec 8 bits on peut représenter $2^8 = 256$ nombres entiers : les nombres de 0 à 255.
Avec 16 bits on peut représenter $2^{16} = 65\,536$ nombres entiers de 0 à 65 535.

Au lieu de faire des raisonnements logiques sur les informations qu'il reçoit, le microprocesseur d'un ordinateur opère des calculs sur les bits qui lui sont transmis par le **bus de données**, donc des calculs avec des 0 et des 1. Mais il ne calcule jamais sur 1 seul bit d'information à la fois, mais plutôt sur un ou plusieurs **octets** en même temps.

Définition : L'octet.

En informatique, ce qu'on appelle un **octet** est un mot **information** de longueur **8 bits**.
L'**octet** représente un mot (raison historique) : **1 octet = 8 bits = x x x x x x x x**

Remarques :

Actuellement, nos ordinateurs travaillent sur 64 bits, soit 8 octets à la fois pour les données et les adresses mémoire.
On trouve parfois le **Byte** comme unité d'information :

1 Byte = 1 B = 1 octet = 8 bits.

Multiples :

1 ko = 2^{10} octets = 1 024 octets
1 Mo = 2^{10} ko = 1 024 $\times$ 1 024 = $1\,024^2$ octets = 1 048 576 octets
1 Go = 2^{10} Mo = 1 024 Mo = $1\,024^3$ octets soit plus d'un milliard de mots
Les disques durs actuels embarquent de l'ordre d'1 To de mémoire, soit 2^{40} octets.

Remarque : Aujourd'hui on utilise 1 ko = 1 000 octets, 1 Mo = 1 000 ko et 1 Go = 1 000 Mo.

Exercice :

- Regardez la mémoire sur votre ordinateur dans les propriétés système : disque dur, ram, architecture du cpu et système d'exploitation.
- Notez ces informations et recherchez leur signification si nécessaire.

Exercice : Un processeur effectue des calculs en nombres entiers codés sur 32 bits.

- De combien de nombres entiers disposera-t-il pour les calculs ?
- Quel est le plus grand entier positif disponible ?
- Comment pourrait-on disposer également d'entiers négatifs ?
- Quel serait alors l'intervalle des nombres disponibles ?

Exercice :

1) Peut-on effectuer les additions suivantes sur 8 bits ? $97_{10} + 31_{10}$ et $136_{10} + 123_{10}$.

2) Effectuer ces additions binaire et vérifier les résultats décimaux.

Définition : Le logarithme de base 2.

Le logarithme de base 2 d'un nombre entier n est égal, en première approximation, au nombre de fois où l'on peut diviser l'entier n par 2.

Ce nombre s'écrit $\log_2(n)$.

Exemples : $\log_2(1)=0$, $\log_2(2)=1$, $\log_2(4)=2$, $\log_2(8)=3$ et $\log_2(256)=8$.

Remarques :

En réalité, la fonction mathématique logarithme de base 2 est utilisable pour tous les nombres réels positifs, si bien qu'on peut toujours calculer $\log_2(x)$.

```
>>> from math import log2, pi
>>> log2(1024)
10.0
>>> log2(2**13)
13.0
>>> log2(255)
7.994353436858858
>>> log2(pi)
1.6514961294723187
```

Le résultat est toujours un nombre à virgule flottante en python.

Mais en mathématiques, le résultat est un entier si le nombre est une puissance de 2.

- En effet, la fonction $\log_2()$ vérifie la propriété suivante : $\log_2(2^n)=n$.
- Donc le nombre $\log_2(2^n)$ correspond au nombre de fois où l'on peut diviser 2^n par 2.
- Si un entier m n'est pas une puissance de 2, on prendra la partie entière de $\log_2(m)$ pour savoir combien de fois m est divisible par 2 : $\log_2(1000)$ a pour partie entière 9, donc 1 000 est 9 fois divisible par 2.
- Avec la calculatrice, $\log_2(n)$ se tape $\ln(n)/\ln(2)$.

Propriété : Le nombre de bits nécessaires pour représenter un entier n en binaire est égal à :

$$1 + \text{partie entiere de} \log_2(n)$$

Pour des valeurs de n suffisamment grandes, ce nombre est équivalent à $\log_2(n)$

Exemple :

10 000 est 13 fois divisible par 2 car $\log_2(10000)=13,28\ldots$,
mais il faut 14 bits pour écrire ce nombre en binaire :

$$10\,000_{10}=10\,0111\ 0001\ 0000_2$$

3) Représentation des entiers relatifs en binaire :

Un **mot** informatique d'un **octet** permet de représenter 256 objets différents, donc par exemple tous les entiers de 0 à 255 , ou encore de 0 à 2^8-1 puisque $2^8=256$.

En effet, le nombre décimal 255_{10} correspondant au binaire composé de 8 fois le bit 1 :

$$255_{10}=1111\,1111_2=1\,0000\,0000_2-1_2=2^8-1\,.$$

Avec 2 octets ou 16 bits, on obtient l'intervalle de nombres entiers $\left[0\,,\,65\,535\right]$.

Avec 3 octets ou 32 bits, on obtient l'intervalle $\left[0\,,\,2^{32}-1\right]$ soit un peu plus de quatre milliards de nombres entiers.

Un nombre codé sur n bits admet 2^n valeurs différentes possibles. Si l'on ne prend que des entiers positifs, cela représente l'intervalle $\left[0\,,\,2^n-1\right]$.

Passage aux entiers négatifs :

Décidons maintenant de coder nos entiers relatifs *sur 4 bits* pour commencer.

Pour décider du **signe d'un nombre binaire**, on peut convenir de définir un **bit de signe**. Choisissons le premier de poids fort pour représenter le signe de nos entiers:

MSB = 0 pour les entiers positifs
MSB = 1 pour les entiers négatifs.

Avec cette convention, an aura par exemple :

$6_{10}=0110_2$ et $-6_{10}=1110_2$ en codage sur 4 bits.

Le plus grand nombre sera donc $0111_2=7_{10}$ et le plus petit $1111_2=-7$.
Il y aura aussi deux zéros: 0000_2 et 1000_2 .

Exercice : En codant les décimaux 7 et -5 de cette manière, vérifier s'il est possible de faire une soustraction.

Première tentative de résolution du problème : Le $CA1_n$, complément à un sur n bits.

Définition : Le **complément à 1 sur n bits** d'un décimal se note $CA1_n$.

- Le $CA1_n$ d'un décimal positif est sa représentation binaire classique sur n bits
- Le $CA1_n$ d'un décimal négatif s'obtient en inversant les n bits de la représentation binaire du même décimal positif :

$+6$ s'écrira 0110 en $CA1_4$

-6 s'écrira 1001 en $CA1_4$

Exemple :

Le $CA1_8$ du décimal 100 est $0110\ 0100_2$, et celui de -100 est donc $1001\ 1011_2$.
Le $CA1_8$ du décimal 155 est $1001\ 1011_2$, et celui de -155 est donc $011\ 0100_2$.

On voit donc qu'il devient difficile de différencier les entiers positifs des entiers négatifs.

De plus, $100-100=0$ doit correspondre à l'opération binaire :

$$0110\ 0100_2+1001\ 1011_2=1111\ 1111_2=0_{10}$$

Ce qui fait qu'il existe à nouveau deux zéros binaires : $0000\ 0000_2$ et $1111\ 1111_2$.
Il faut donc effectuer deux tests au lieu d'un pour savoir si un résultat est nul.

Propriété : Sens du $CA1_n$.

Si un décimal k_{10} est codé en binaire en complément à 1 sur n bits,
alors on doit avoir :

$$k_{10}+CA1_n(k_{10})=\underbrace{111\ldots1_2}_{n \text{ fois le chiffre } 1}=(2^n-1)_{10} \text{ et donc } CA1_n(k_{10})=(2^n-1)_{10}-k_{10}$$

Exemples :

- Le $CA1_4$ du décimal 5 est le décimal $(2^4-1)-5=15-5=10$ qui se code 1010_2.
- Le $CA1_8$ du décimal 100 est le décimal $(2^8-1)-100=255-100=155$.
- Le $CA1_8$ du décimal 127 est le décimal $255-127=128$.

Exercice : Caluler $7-5$ puis $5-7$ en $CA1_4$. Cela fonctionne-t-il bien ?

Deuxième tentative de résolution du problème : Le $CA2_n$, complément à deux sur n bits.

On décide du nombre de bits utilisés pour coder les entiers : prenons **8 bits** ici.

Le **MSB** est utilisé pour le signe des entiers : **0 ==> positif** et **1 ==> négatif**.

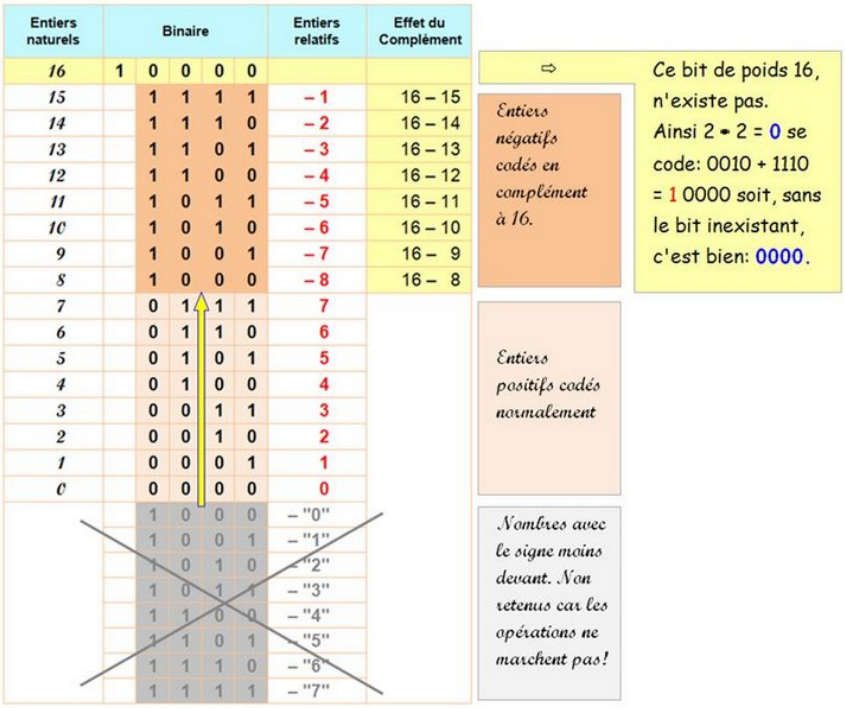
Il reste donc **7 bits** pour coder les **décimaux positifs** et le **zéro**, soient les 128 décimaux de 0 à 127, codés de $0000\ 0000_2$ à $0111\ 1111_2 = 2^7 - 1 = 128 - 1 = 127_{10}$.
En complément à deux, on continue à représenter **les entiers positifs comme avant, avec un MSB = 0 :** $0xxx\ xxxx$ pour coder les décimaux de 0 à $2^7 - 1 = 127_{10}$.

Il reste également **7 bits** pour coder les **décimaux strictement négatifs**, soient les 128 décimaux de -1 à -128. Leur codage en complément à deux suit le principe suivant :

> Après le plus grand nombre positif $0111\ 1111_2 = 127$, on commence la liste des nombres négatifs toujours dans l'ordre croissant mathématique :
>
> $1000\ 0000_2 = -128_{10}$, $1000\ 0001_2 = -127_{10}$ et enfin $1111\ 1111_2 = -1_{10}$.

En complément à deux, les entiers **négatifs** s'écriront sous la forme $1xxx\ xxxx$, **avec un MSB = 1** pour coder les décimaux de -128 à -1.

On représentera désormais **les entiers négatifs par leur complément à deux**.



Cela revient à prendre les nombres d'au-dessus au lieu de ceux d'en-dessous.

Exemples :

- Ainsi, le décimal 1_{10} se code $0000\ 0001_2$ en $CA2_8$, et le décimal -1_{10} se code $1111\ 1111_2$ en $CA2_8$.

  On a donc $1_{10}-1_{10}=0000\ 0001_2+1111\ 1111_2=1\ 0000\ 0000_2=2^8=256_{10}$,
  qui est un nombre qui ne peut pas être codé sur 8 bits.

  Donc $1_{10}+CA2_8(1_{10})=2^8$.

- De même, le décimal $2_{10}=0000\ 0010_2$ et le décimal $-2_{10}=1111\ 1110_2$ en $CA2_8$.
  On a donc encore $2_{10}-2_{10}=0000\ 0010_2+1111\ 1110_2=1\ 0000\ 0000_2=2^8{}_{10}$.

  Donc $2_{10}+CA2_8(2_{10})=2^8$.

Propriété : Sens du $CA2_n$.

Si un décimal k_{10} est codé en binaire en complément à 2 sur n bits,
alors on doit avoir :

$$k_{10}+CA2_n(k_{10})=2^n{}_{10} \text{ et donc } CA2_n(k_{10})=2^n{}_{10}-k_{10}.$$

Exemples :

Prenons un décimal positif de l'intervalle $[0;127]$ et déterminons son $CA2_8$:

$59_{10}=11\ 1011_2$, et $CA2_8(59_{10})=0011\ 1011_2$ car le nombre est positif.

Alors -59_{10} a pour $CA2_8$ le décimal $2^8-59=256-59=197_{10}$, codé $1100\ 0101_2$.
Donc $CA2_8(-59_{10})=1100\ 0101_2$.
On vérifie bien que $CA2_8(59_{10})+CA2_8(-59_{10})=1\ 0000\ 0000_2=256_{10}$.

Prenons un décimal négatif de l'intervalle $[-128;-1]$ et déterminons son $CA2_8$:

$CA2_8(-68_{10})=256-68=188_{10}$ qui se code $1011\ 1100_2$.
Le $CA2_8$ de 188_{10} est le décimal $256-188=68_{10}$ qui se code $0100\ 0100_2$.
On vérifie bien que $CA2_8(-68_{10})+CA2_8(68_{10})=1\ 0000\ 0000_2=256_{10}$.

Remarque :

Comme on a $CA1_n(k_{10}) = (2^n - 1)_{10} - k_{10}$,
alors $CA2_n(k_{10}) = (2^n - 1)_{10} - k_{10} + 1_{10}$, et donc $CA2_n = CA1_n + 1$.

Exemple :

Prenons le binaire $0101\ 1100_2 = 92_{10}$.
Son $CA1_8$ est le binaire $1010\ 0011_2$ et son $CA2_8$ est le binaire $CA1_8 + 1 = 1010\ 0100_2 = 164_{10}$.
L'entier négatif -92_{10} se code donc $1010\ 0100_2$ en $CA2_8$.

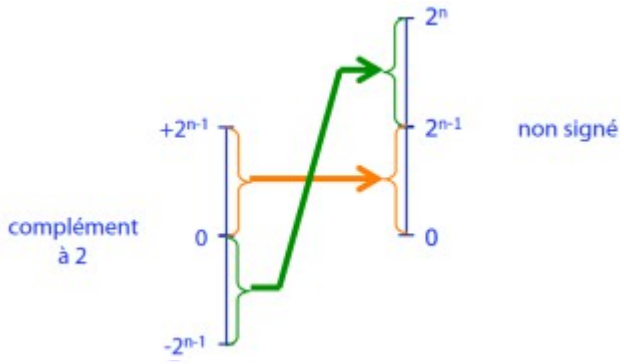
Définition : $CA2_8$ complément à 2 sur 8 bits d'un décimal négatif.

- *méthode 1* pour obtenir le complément à 2 :

$$CA2_8 = CA1_8 + 1_2$$

- *méthode 2* pour obtenir le complément à 2 :

du LSB au MSB :
on inverse tous les bits **après** le premier 1 rencontré



A retenir :

Si l'on décide d'écrire les entiers sur 8 bits, alors on pourra représenter tous les **entiers relatifs** de l'intervalle $[-128\ ;\ 127]$ ou encore $[-2^7\ ;\ 2^7 - 1]$.

Pour $k = 8$ bits, on a les entiers signés :

−128	=	1	0	0	0	0	0	0	0
⋮									
−1	=	1	1	1	1	1	1	1	1
0	=	0	0	0	0	0	0	0	0
1	=	0	0	0	0	0	0	0	1
⋮									
127	=	0	1	1	1	1	1	1	1

En décidant de coder les **entiers signés** sur **n bits**, il est alors possible de représenter tous les entiers de l'intervalle $[-2^{n-1}\ ;\ 2^{n-1} - 1]$.

Par exemple, sur 3 bits on peut représenter les $2^3 = 8$ nombres de -4 à 3.
En enlevant le bit de signe, il reste $2^2 = 4$ nombres posibles.
Pour les positifs, les quatre entiers de 0 à $3 = 2^2 - 1$ qui s'écrivent de 000_2 à 011_2.
Pour les négatifs, les quatre entiers de -4 à -1 qui s'écrivent de 100_2 à 111_2.

Remarques :

- En Python, les nombres entiers peuvent avoir n'importe quelle taille : Python3 adapte la taille du codage des nombres en fonction de ses besoins.
- Le CA2 de 0 est égal à 0 en ne prenant pas en compte la retenue.
- Le CA2 du CA2 d'un **entier signé** est l'entier lui-même.

Exemples de calculs en CA2 : On considère les décimaux positifs $a = 91_{10}$ et $b = 48_{10}$.

1) Calculons $a_{10} + b_{10} = 139_{10}$ en binaire :

Nous voyons dores et déjà que nous allons dépasser la capacité de représentation des entiers signés sur 1 octet, puisque le résultat est plus grand que le plus grand nombre positif possible qui est $0111\,1111_2 = 2^7 - 1 = 127$. Voyons cela :

$a = 91_{10} = 0101\,1011_2$ et $b = 48_{10} = 0011\,0000_2$ en $CA2_8$.

```
      +1 dernier report
        0 1 0 1   1 0 1 1
  +     0 0 1 1   0 0 0 0
  = 0   1 0 0 0   1 0 1 1
retenue
```

Le bit de parité montre un binaire signé négatif, ce qui n'est pas possible. Il y a un dépassement de capacité appelé **Overflow**.

2) Calculons maintenant $a_{10} - b_{10} = 43_{10}$ en binaire :

$a = 91_{10} = 0101\,1011_2$ et $-48_{10} = 1101\,0000_2$ en $CA2_8$.

```
      +1 dernier report
        0 1 0 1   1 0 1 1
  +     1 1 0 1   0 0 0 0
  = 1   0 0 1 0   1 0 1 1
retenue  bit de signe
```

En jetant la retenue à la poubelle, le résultat est le binaire positif $0010\,1011_2 = 43_{10}$.
Cela fonctionne.
Il n'y a pas d'Overflow dans ce cas !

3) Calculons $b_{10} - a_{10} = -43_{10}$ en binaire :

$b = 48_{10} = 0011\ 0000_2$ et $-91_{10} = 1010\ 0101_2$ en CA2

		0 dernier report							
		0	0	1	1	0	0	0	0
+		1	0	1	0	0	1	0	1
=	0	1	1	0	1	0	1	0	1
	retenue								

Le résultat obtenu est le binaire négatif $1101\ 0101_2$ représenté en CA2, sans overflow.
Son CA2 est $0010\ 1011_2 = 43_{10}$.

Un résultat négatif est interprété par le processeur en représentation en CA2.
Pour obtenir la valeur absolu du résultat il faut soit prendre le CA2 du binaire obtenu, soit enlever 256 au résultat décimal si l'on travaille sur 8 bits.

4) Calculons pour finir $-a_{10} - b_{10} = -139_{10}$ en binaire :

$-91_{10} = 1010\ 0101_2$ et $-48_{10} = 1101\ 0000_2$ en CA2.

		0 dernier report							
		1	0	1	0	0	1	0	1
+		1	1	0	1	0	0	0	0
=	1	0	1	1	1	0	1	0	1
	retenue								

Le résultat obtenu positif est impossible.
Dans ce cas encore il y a **Overflow**.

Remarque : C'est à nous de décider comment le processeur doit réagir en cas d'overflow.

Exercices : Soustractions sur 8 bits avec ou sans retenue, et reconnaitre l'overflow.

A retenir :

- On lit le résultat en CA2 dans tous les cas.
- Il y a **overflow** lorsque **la retenue est différente du dernier bit de report**.
- On peut tester cela en plaçant une porte XOR sur les 2 derniers carry out.
- Pour étendre un nombre signé on rajoute à sa gauche son bit de signe.

Algorithmes : conversions binaire vers $CA2_8$, puis décimal vers $CA2_8$.

4) Représentation des réels en nombre à virgule flottante :

En base 10 on a : $452\ ,\ 786 = 400 + 50 + 2 + 0{,}7 + 0{,}08 + 0{,}006$

$= 4\times100 + 5\times10 + 2\times1 + 7/10 + 8/100 + 6/1000$

$= 4\times10^{2} + 5\times10^{1} + 2\times1 + 7\times10^{-1} + 8\times10^{-2} + 6\times10^{-3}$

En base 2, on aura de manière analogue :

$1010\ ,\ 101 = 1\times2^{3} + 0\times2^{2} + 1\times2^{1} + 0\times2^{0} + 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3}$

$= 8 + 0 + 2 + 0 + 0{,}5 + 0 + 0{,}125$

$= 10\ ,\ 625$

On peut démontrer rigoureusement que tout nombre réel positif inférieur à 1 peut s'écrire sous une telle forme. Reste à choisir une convention pour le signe.

Dans le binaire 1010 , 101 : 1010 est la **partie entière**
101 est la **partie fractionnaire**

Exercice : Donner la valeur décimale des nombres $1100\ ,\ 011_2$ et $10010\ ,\ 110_2$.

Exemple du calcul inverse : traduire en binaire le nombre 78,347 décimal

Partie entière : 78

Nous opérons une suite de divisions par 2 et retenons les divers restes.
Ces restes sont repris à l'envers

78	2
0	39 | 2
1	19 | 2
1	9 | 2
1	4 | 2
0	2 | 2
0	1

Résultat : 1001110

Partie fractionnaire : 0,347

Voici comment on peut procéder :

0,347 . 2 = 0,694 < 1 je pose 0 : 0,347 = 0,0...
0,694 . 2 = 1,388 > 1 je pose 1 : 0,347 = 0,01...
0,388 . 2 = 0,766 < 1 je pose 0 : 0,347 = 0,010...
0,766 . 2 = 1,552 > 1 je pose 1 : 0,347 = 0,0101...
0,552 . 2 = 1,104 > 1 je pose 1 : 0,347 = 0,01011...
0,104 . 2 = 0,208 < 1 je pose 0 : 0,347 = 0,010110...
0,208 . 2 = 0,416 < 1 je pose 0 : 0,347 = 0,0101100...
0,416 . 2 = 0,832 < 1 je pose 0 : 0,347 = 0,01011000...
0,832 . 2 = 1,664 > 1 je pose 1 : 0,347 = 0,010110001...
0,664 . 2 = 1,328 > 1 je pose 1 : 0,347 = 0,0101100011...

Pour plus d'explications cliquer ici : →

Résultat final

78,347 (écrit ici sous forme décimale)
est égal à 1001110,0101100011 écrit en binaire.

Représentation machine des nombres réels :

Les données sont représentées en binaire par des **mots** informatique d'un certain nombre fixé de bits, rarement 8, mais plutôt 16 ou **32 bits** en **simple précision**, et **64 bits** en **double précision**.

En 16 bits par exemple, on peut décider de prendre 1 octet à gauche pour la partie entière, et 1 octet à droite pour la partie fractionnaire: xxxx xxxx , xxxx xxxx
C'est la représentation en **virgule fixe**.

Mais comme elle comporte trop d'inconvénients, les processeurs utilisent aujourd'hui la représentation en nombre à **virgule flottante** – Floating Point -

Par exemple, le binaire 0 , 00000001 00101110 ne pourrait pas être représenté en virgule fixe sur 2 octets, car sa partie fractionnaire dépasse l'octet.

Virgule flottante :

Pour comprendre la représentation en virgule flottante il faut déjà savoir écrire un nombre en **notation scientifique** de la forme $n \times 10^p$ où $n \in [1 ; 10[$ et $p \in \mathbb{Z}$.
On peut déjà remarquer que le nombre 0 ne peut pas s' écrire avec ce format.

Exercice : Ecrire les décimaux suivants en notation scientifique:

$a = 0,00000123$ $b = -45600$ $c = 7809,00604$ $d = -0,00050087$

Définition : Signe, Mantisse et Décalage algébrique d'un nombre réel

Tout nombre réel peut se définir sous une forme utilisant S + D + M où on a:

S est le **Signe** du nombre: +/- en décimal (codé 0 ou 1 en binaire)
D est le **Décalage algébrique** qui correspond à la position de **l'exposant** dans **l'écriture scientifique**, et qui posséde donc un signe
M est la **Mantisse**, qui correspond aux **chiffres significatifs**

Exemple : - 0 , 000805002 = - 8, 05002 $\times$ 10^{-5} donc dans ce cas on aura:
S = - D = - 5 M = 8 , 05002

En fonction de la taille que l'on se fixera pour représenter les Décalages et la mantisse, nous disposerons de plus ou moins de nombres réels, et donc de plus ou moins de précision.

La Mantisse **M** possède une partie entière et une **partie fractionnaire F**.

Transposons cette procédure à l'écriture binaire :

Nb = - 100011 , 101011

S = - (codé 1) D = +5 (codé en CA2) M = 1 , 00011101011

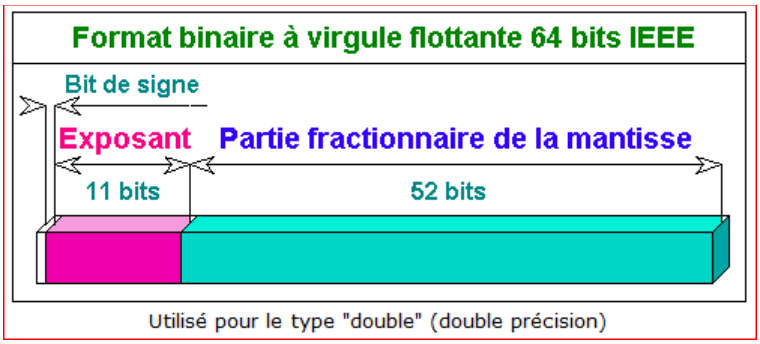
donc Nb = - 1 , 00011101011 $\times$ 2^{+5}

Exercice : Signe, Déplacement algébrique et Mantisse

1) Déterminer les valeurs de S, D et M pour les binaires suivants, sur 16 bits :
   + 0 , 00010101 et – 0 , 01001011

2) Que remarque-t-on concernant la mantisse de ces binaires ?

Norme IEEE : La représentation binaire des nombres réels en double précision utilise un format sur 64 bits qui ne correspond pas à ce qui précède :



La mantisse M d'un nombre binaire commence toujours par 1, et ce premier bit est ignoré dans la norme IEEE 754.

L'exposant utilisé dans la norme IEE754 n'est pas l'exposant de décalage de la virgule, comme calculé plus haut pour former la mantisse et appelé D.

L'implémentation de la norme utilise un **exposant positif E défini sur un nombre n de bits fixés**, et peut donc prendre toutes les valeurs de 0 à 2^n-1 .

Comme cet exposant E dépend du Déplacement algébrique D de la notation scientifique, il faut opérer un décalage B (pour Bias en anglais).

On écrit alors $E = D + B$.

Et pour obtenir des décalagees algébriques en nombres binaires signés sur n bits dans l'intervalle centré $\left[-\left(2^{n-1}-1\right);2^{n-1}\right]$, cela impose de prendre:

$E_{max}=2^n-1=D_{max}+B=2^{n-1}+B$ et donc le décalage vaut $B=2^{n-1}-1$.

On aura alors :

Schéma de représentation des nombres en virgule flottante et en représentation binaire

$$(signe)\times 2^{exposant-décalage}\times 1,mantisse$$

Exemples :

1) Codons le nombre décimal a = - 352 , 0 en simple précision sur 32 bits.

Valeur absolue: $352_{10}=0001\ 0110\ 0000_2=0x160$

Mantisse = $0001\ 0110\ 0000_2=1\ ,\ 01100000\times 2^8$

Partie fractionnaire de la mantisse sur 23 bits F = 011 0000 0000 0000 0000 0000

E = D + B = 8 + 2^{8-1} - 1 = 8 + 127 = 135 codé $1000\ 0111_2$ sur 8 bits

Ce qui donne la représentation norme IEEE754 suivante :

1100 0011 1011 0000 0000 0000 0000 0000 = 0x C3 B0 00 00

Le décimal a = - 352 s'écrit alors: 1100 0011 1011 0000 0000 0000 0000 0000.

2) Codons le nombre décimal b = - 2019 , 28125

Valeur absolue

partie entière = 2019_{10} = $0000\ 0111\ 1110\ 0011_2$

partie fractionnaire F = $0\ ,\ 28125_{10}$

0 ,28125×2 = 0,5625 donc 0

0 ,5628×2 = 1,125 donc 1

0 ,125×2 = 0,25 donc 0

0 ,25×2 0,5 donc 0

0 ,5×2 = 1 donc 1

0×2 = 0 donc plus que des 0

Alors $b_{10}=-0111\ 1110\ 0011\ ,\ 01001=-1\ ,\ 111110001101001\times 10^{10}$

Partie fractionnaire de la mantisse sur 23 bits F = 111 1100 0110 1001 0000 0000
E = D + B = 10 + 127 = 137_{10} = $1000\,1001_2$

Le décimal b = - 2019 , 28125 s'écrit donc :

1100 0100 1111 1100 0110 1001 0000 0000 = 0x C4 FC 69 00

Exercice :

1) Convertir au format IEEE754 le décimal c = 0 , 2019.

2) Quelle est l'erreur commise en utilisant sa représentation binaire ?

Correspondance Exposant – déplacement :

E	0	1	...	B (127)	B + 1 (128)	...	2^n-2 (254)	2^n-1 (255)
D (sur 8 bits)	- B (-127)	- B + 1 (-126)	...	0	1	...	$2^{n-1}-1$ (127)	2^{n-1} (128)
	128 valeurs $D \leqslant 0$				128 valeurs $D > 0$			

Les deux colonnes grisées sont réservées à des exceptions :

Le **zéro** sera représenté par un exposant $E = 0$ et une partie fractionnaire $F = 0$.
Il y a donc deux zéros à cause du bit de signe.

Tous les autres nombres ayant un exposant $E = 0$ et une partie fractionnaire $F \neq 0$ seront déclarés hors norme, ou nombres **dénormalisés**.

Pour mémoriser un résultat infini ou une impossibilité de calcul, on réserve les nombres de la dernière colonne, correspondant à $E = E_{max} = 2^n - 1 = 1111\,1111_2 = 255$ sur 8 bits.

Parmi eux, les nombres infinis (**INFINITY**) correspondent à $E = 2^n - 1$ et $F = 0$, tandis que les nombres vérifiant $E = 2^n - 1$ et $F \neq 0$ sont des non-nombres, notés **NaN**, ce qui signifie **Not a Number**.

Remarques :

Les nombres **INFINITY** permettent d'enregistrer le résultat telle qu'une division par 0, et les **NaN** des erreurs de saisie par exemple.
Les nombres **dénormalisés** ont un exposant $E = 0$ et une partie fractionnaire $F \neq 0$.

Il faut les interpréter comme ayant une **partie entière nulle** et une **partie fractionnaire non nulle**. Pour assurer la continuité avec les nombres normalisés, on retiendra un déplacement algébrique décalé d'un rang vers la gauche:

$D = -B + 1$ et donc $D = -126$ sur 8 bits.

Ces nombres s'écriront donc +/- 0 , F $\times 2^{-126}$

Exercice : Déterminer les valeurs absolues maximale et minimale d'un nombre dénormalisé.