None

In [1]:
```
mode="PDF" # Autre possibilités : "PDF+SOL" (solutions), "PROF" (interactif+solutions), "INTER" (interactif)
cle=1 # Modifiez cette ligne pour changer les données de certains exercices
```

In [2]:

```
Ce cours a été régénéré le 2020-09-20 23:22:37.745451. Mode sans corrigé. Mode statique. Modèle numéro 1.
```

# Les nombres

Un *nombre* est un objet (abstrait) qui admet de nombreuses représentations. L'idée de quantité et une représentation visuelle précèdent sans doute l'écriture *(unaire)*. Un jeu de règles de représentation des nombres sous forme de signes écrits est un système de numération.

On représente aussi les nombres sur d'autres *supports* que l'écrit: représentations par sons, par objets (nombre de bougies sur un gâteau). Cela ne change pas le nombre (information), 27 bougies représentent bien 27 éléments (années écoulées, ici) autant que « 2 » collé à « 7 », ou que XXVII (chiffres romains).

## Écrire les nombres

### La numération positionnelle

> **Système de numération positionelle :** Un ensemble fini de symboles B (appelés chiffres) auxquels est associé une valeur entière de 0 à B − 1, où B est le nombre d'éléments de B. B est la *base*. La valeur d'une suite finie de k chiffres $\alpha_{k-1}\alpha_{k-2}\cdots\alpha_0$ est la somme: $\alpha_{k-1}B^{k-1} + \cdots + \alpha_1 B + \alpha_0 = \sum_{i=0}^{k-1}\alpha_i B^i$.

Le mot *chiffre* vient de l'arabe الصِّفْر ṣ-ṣifr et désignait le zéro.

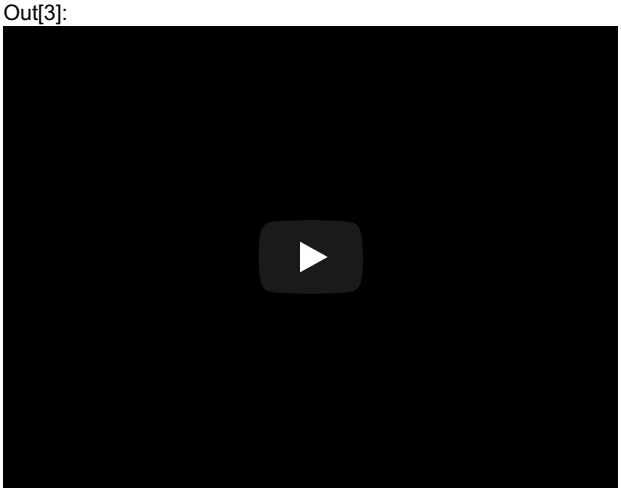
> **Exemple :** en base 5: le nombre $132_5$ vaut $1 \times 5^2 + 3 \times 5^1 + 2 \times 5^0$, soit $25 + 15 + 2 = 42_{10}$.

$B^i$ est le *poids* du i-ème chiffre (en comptant à partir de 0 et de la droite).

In [3]:

### Activité : Les shadoks

Out[3]:



### Les autres systèmes

- Systèmes de numération additifs (chiffres grecs, égyptiens): ∩ ∩ | | | | | | |, par exemple. Chaque poids est représenté par un symbole distinct, la position n'est pas importante. À un détail près, les chiffres romains le sont aussi.
- Systèmes hybrides (numérotation chinoise ou japonaise, français) : on utilise des chiffres fixes, mais on intercale un symbole (écrit) ou un mot (oral) différent pour chaque poids.
- Des systèmes de numération exotiques: les poids ne sont pas 1, B, $B^2$, $B^3$, etc. mais les valeurs d'une suite (strictement croissante): par exemple, numération de Fibonacci.

### La base 10

Système décimal, utilisé depuis le cinquième siècle en Inde, apporté par les Arabes en Europe dans le X$^{e}$ siècle. Les chiffres sont B = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, et la base B = 10.

> **Exemple :** mille cinq cent quatre-vingt-quatre se représente par $1684_{10}$, qui s'interprète comme
>
> $1 \times 10^3 + 6 \times 10^2 + 8 \times 10 + 4$

## Représenter les nombres en informatique

| Base | Chiffres | Préfixe C/Python | Exemple | Utilisation |
|---|---|---|---|---|
| 2 | {0,1} | 0b | 0b10110 | Codage bas niveau |
| 8 | {0,1,2,3,4,5,6,7} | 0/0o | 026/0o26 | Peu utilisé |
| 10 | {0,1,2,3,4,5,6,7,8,9} | (aucun) | 22 | valeurs décimales |
| 16 | {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} | 0x | 0x16 | Valeurs binaires écrites de façon compacte |

En binaire, un chiffre est appelé un *bit*. Quel hasard !

In [4]:
```
a=0b10110
b=0o26
c=22
d=0x16

if a==b and a==c and a==d:
    print("Tous ces nombres sont égaux à {0}={1}={2}={3} !".format(a,bin(a),oct(a),hex(a)))
else:
    print("a="+str(bin(a)))
    print("b="+str(oct(b)))
    print("c="+str((c)))
    print("d="+str(hex(d)))
```

```
Tous ces nombres sont égaux à 22=0b10110=0o26=0x16 !
```

# Changer de base

## De la base x à la base 10

Il suffit d'utiliser la méthode du recalcul ! On sait faire les opérations en décimal, on applique donc la définition de base.

Par exemple, 0x1D4 = $1 \times 16^2 + 13 \times 16 + 4 \times 1 = 256 + 208 + 4 = 468$

## De la base 10 à la base 2

On peut procéder par divisions successives, ou par soustractions successives. La deuxième méthode est la plus efficace pour le binaire.

**Division successive :** On divise à chaque fois le nombre par la base (ici, 2). Le reste de la division entière est le chiffre suivant du nombre, puis on repart avec le résultat de la division. Quand on est arrivé à 0, c'est fini.

**Soustraction successives :** On prend la plus grande puissance de deux possible, puis à partir de là on note 1 si on peut soustraire le nombre, et 0 sinon. Puis on passe à la puissance de deux juste inférieure, jusqu'à arriver à 1.

In [5]:

### Activité : Conversion de la base 10 vers la base 2

Donner un nombre à décomposer en base 2.

Un entier > `129`

129/2 = 64, reste 1, j'ai pour le moment 1

64/2 = 32, reste 0, j'ai pour le moment 01

32/2 = 16, reste 0, j'ai pour le moment 001

16/2 = 8, reste 0, j'ai pour le moment 0001

8/2 = 4, reste 0, j'ai pour le moment 00001

4/2 = 2, reste 0, j'ai pour le moment 000001

2/2 = 1, reste 0, j'ai pour le moment 0000001

1/2 = 0, reste 1, j'ai pour le moment 10000001

La conversion de 129 en base 2 est 10000001

---

Je peux enlever 128 de 129, il reste 1. Je note donc 1.

Je ne peux pas enlever 64 de 1. Je note donc 10.

Je ne peux pas enlever 32 de 1. Je note donc 100.

Je ne peux pas enlever 16 de 1. Je note donc 1000.

Je ne peux pas enlever 8 de 1. Je note donc 10000.

Je ne peux pas enlever 4 de 1. Je note donc 100000.

Je ne peux pas enlever 2 de 1. Je note donc 1000000.

Je peux enlever 1 de 1, il reste 0. Je note donc 10000001.

Résultat final : 10000001

## De la base 2 à 8/16 et inversement

Entre les bases qui sont des puissances l'une de l'autre (ce qui est le cas pour la base 2 et 16 = $2^4$, ou 2 et 8 = $2^3$, il est possible de prendre des raccourcis par rapport aux méthodes précédentes.

En effet, si on regarde par exemple de 2 à 16 (cas fréquent), un chiffre hexadécimal représente exactement 4 chiffres binaires : dans l'écriture, il suffira de prendre les bits par paquets de 4 (à partir de l'unité, pas dans l'autre sens) pour obtenir le chiffre hexadécimal correspondant.

De la même façon, dans l'autre sens, un chiffre hexadécimal est remplacé par exactement 4 bits (3 dans le cas de l'octal).

Ça correspondrait au calcul suivant : 0x5D = 5 × 16 + 13 × 1 = (1 × 4 + 1 × 1) × 16 +
(1 × 8 + 1 × 4 + 1 × 1) = $1 \times 2^6 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0$ = 0b01011101

Il faut connaître par cœur la table de correspondance suivante :

| Binaire | Hexadécimal | Décimal |
|---|---|---|
| 0000 | 0 | 0 |
| 0001 | 1 | 1 |
| 0010 | 2 | 2 |
| 0011 | 3 | 3 |
| 0100 | 4 | 4 |
| 0101 | 5 | 5 |
| 0110 | 6 | 6 |
| 0111 | 7 | 7 |
| 1000 | 8 | 8 |
| 1001 | 9 | 9 |
| 1010 | A | 10 |
| 1011 | B | 11 |
| 1100 | C | 12 |
| 1101 | D | 13 |
| 1110 | E | 14 |
| 1111 | F | 15 |

Pour l'octal, il suffit de prendre la première moitié du tableau et d'enlever le premier 0 du nombre hexadécimal.

In [6]:

Activité : Changement de bases

Convertissez 217 en base 2 > _____

Convertissez 0b10001011 en base 10 > _____

Convertissez 0xa0 en base 10 > _____

Convertissez 1127 en base 16 > _____

Convertissez 0x174 en base 2 > _____

Convertissez 0b1111010010110 en base 16 > _____

# Entiers relatifs et réels

## Définitions

Les entiers relatifs existent évidemment dans toutes les bases. Il suffit de rajouter un bit d'information pour avoir le signe. La convention de notation est calquée sur le décimal : on rajoute un signe `-` devant le nombre (en programmation, on le rajoute devant le préfixe).

Les algorithmes de transformation sont quasi-identiques : on travaille sur la valeur absolue du nombre, puis on rajoute le signe après.

Pour les nombres réels, il suffit de rajouter une virgule. Attention, les poids après la virgule ne sont pas 0, 1, 0, 01, etc. mais $B^{-1}$, $B^{-2}$ et la définition ajustée est donc :

> La valeur d'une suite finie de k + n chiffres $\alpha_{k-1}\alpha_{k-2}...\alpha_0, \alpha_{-1}...\alpha-n+1\alpha_{-n}$ est la somme: $\alpha_{k-1}B^{k-1} + \cdots + \alpha_1 B + \alpha_0 + \alpha_{-1}B^{-1} + \cdots + \alpha_{-n}B^{-n} = k-1\sum i=-n\alpha_i B^i$.
>
> **Remarque :** En anglais, on utilise le point `.` comme séparateur au lieu de la virgule.
>
> **Remarque 2 :** De manière générale, on n'utilise pas la notation classique des nombres à virgules dans d'autres bases que 10. On verra plus loin les nombres flottants qu'on utilise à la place.
>
> **Exemple :** En base 2, $-0b111,01 = -7,25$. En base 16, $0xF,E = 15,875$.

## Conversions

Une fois le problème du signe mis de côté (littéralement : on travaille sur les valeurs absolues, puis on rajoute le signe après), ma méthode de conversion pour les valeurs réelles est calquée sur la méthode pour les entiers. On convertit d'abord la partie entière, puis la partie décimale, et on juxtapose les deux.

La méthode des divisions successives devient la méthode des multiplications successives pour la partie fractionnelle : au lieu de diviser par la base et de noter le reste à chaque fois, on multiplie par la base B et on note (et soustrait) la partie entière à chaque fois.

L'explication du processus est simple : si un nombre en base B s'écrit $x = 0, \alpha_{-1}\alpha_{-2}...\alpha_{-n}$, alors

$B \times x = B-1\sum i=-n\alpha_i B^i = -1\sum i=-n\alpha_i B^{i+1} = 0\sum i=-n+1\alpha_{i+1}B^i = \alpha_{-1} + -1\sum i=-n+1\alpha_{i+1}B^i$. La partie entière est donc exactement le premier chiffre après la virgule.

> **Exemple :** En décimal, si on multiplie 0,351 par 10, on obtient d'abord 3,51 (donc 3 est le premier chiffre après la virgule), on enlève le 3 pour obtenir 0,51 et on recommence : 0,51 fois 10 donne 5,1, donc le deuxième chiffre est 5, etc. En binaire, on multiplie par 2.

In [7]:

Activité : Changement de bases

Convertissez 12,5 de la base 10 à la base 16 > _____

Convertissez 0x1b,a de la base 16 à la base 2 > _____

Convertissez 0b10010,101 de la base 2 à la base 10 > _____

Convertissez 0x53,a de la base 16 à la base 10 > _____

Convertissez 42,125 de la base 10 à la base 2 > _____

Convertissez 0b111101101,1 de la base 2 à la base 16 > _____

# Représentation des entiers

## Qu'est-ce qu'un codage ?

Les ordinateurs ne disposent pas d'une mémoire infinie. Le repérage dans la mémoire se fait en utilisant des *cases mémoires* qui peuvent stocker une quantité d'information limitée (tout le temps la même quantité).

Pour des raisons d'organisation, il n'est pas non plus souhaitable de stocker les données, surtout les plus nombreuses, sur une taille qui varie. Cela pose des problèmes pour les déplacer, les copier, et même les consulter (par exemple, si on range une dizaine de données à la suite les unes des autres, consulter la neuvième peut obliger à lire toutes celles qui précèdent).

Pour toutes ces raisons, on transforme les données dans un format qui autant que possible utilise une taille fixe, connue à l'avance. Elle est d'autant plus fixe que les données sont élémentaires : la gestion des nombres entiers, par exemple, se fait en partie directement sur des circuits électroniques spécialisés (les *registres*) qui ne sont pas extensibles.

On va donc procéder à une transformation des données, qui sans changer le concept associé, va permettre d'avoir des données qui sont toutes intelligibles de la même façon et avec une taille commune. C'est opération est le *codage*, et ne serait rien sans l'opération inverse de *décodage*.

Il est à noter que le *codage* ne vise pas à cacher l'information ou à la rendre illisible pour quelqu'un : les règles de codage doivent au contraire être les plus simples possibles, totalement non-ambiguës ; en d'autres termes, un ordinateur doit être capable de le faire.

Le choix d'un codage peut par contre imposer des contraintes sur les concepts que l'on veut coder; en particulier des limites sur ce que l'on peut coder et ce que l'on ne peut pas. Par exemple, si on a un seul bit d'information, on ne pourra pas coder plus que deux données de même type différentes. Et pire, si on a un seul bit d'information, il y aura forcément des données de types différents qui auront un codage identique.

## Codage des entiers

Les premiers objets que l'on peut coder sont les nombres; et en particulier les entiers.

Le codage des entiers positifs est le plus simple à appréhender : on utilise l'écriture binaire du nombre, et pour atteindre la taille désirée, on complète par des zéros (à gauche, comme ça il n'y a pas d'ambiguïté sur le décodage). Ce codage s'appelle le *codage naturel* ($NAT_{(n)}$).

Évidemment, il ne permet pas de coder des nombres négatifs, ni des nombres trop grands. Les limites du codage $NAT_n$ sont les nombres entiers de 0 à $2^n - 1$. Pour des raisons de compacité on note souvent les résultats non pas en binaire, mais en hexadécimal.

**Exemple :** On veut coder 521 en NAT 12 bits. 521 = 0b1000001001, donc $521NAT_{12}$ = 001000001001 = 0x209.

Pour coder des nombres à la fois négatifs et positifs, il existe trois codages différents : VA+S, C1, C2. Le codage C2 est celui qui est utilisé en machine pour coder les nombres entiers qui peuvent être aussi bien négatifs que positifs. les limites du codage $C2_{(n)}$ sont de $-2^{n-1}$ à $2^{n-1} - 1$.

L'algorithme de codage en C2 est le suivant :

1. Si le nombre est positif, on utilise le codage $NAT_{(n)}$
2. Si le nombre est négatif, on utilise le codage $NAT_{(n)}$ pour $|x| - 1$, puis on *complémente* tous les bits (c'est-à-dire qu'on change les 1 en 0 et inversement).

**Exemple :** On veut coder -28 en C2 8 bits. $|-28| - 1 = 27 = 0b11011NAT_8 = 00011011$. Donc $-28C2_8 = 11100100 = 0xE4$.

In [8]:

### Activité : Codage des entiers

Coder en NAT 8 bits écrit en hexadécimal la valeur 11 (en décimal) > _____

Coder en C2 8 bits écrit en hexadécimal la valeur -61 (en décimal) > _____

Coder en C2 12 bits écrit en hexadécimal la valeur -639 (en décimal) > _____

Coder en NAT 16 bits écrit en hexadécimal la valeur 29057 (en décimal) > _____

## Codage des réels

Les réels doivent être codés aussi. Un codage simple serait d'indiquer la position de la virgule. C'est la notation dite *à virgule fixe*, où la virgule est toujours à côté de l'unité.

Toutefois, les échelles de valeurs des réels sont bien plus étendues que les entiers, et les codages ne sont pas adaptés. On utilise donc un dérivé de la notation scientifique, nommé la *virgule flottante* — parce que la virgule est à côté d'un nombre dont le poids n'est pas fixe.

La définition est la suivante : tout nombre x non nul peut s'écrire

$x = (-1)^s \times m \times B^e$ avec $1 \leqslant$
En base 10, c'est précisément la notation spécifique.

Pour faire la représentation proprement dite, il suffit de connaître combien de place on réserve à l'écriture de s, de m et de e (parce que $B=2).

Le codage choisi est le suivant (sur 32 bits) :

On réécrit x comme valant x=(-1)^s\times 1,M\times B^{E-127} avec M la partie fractionnaire de la valeur m écrit sur 23 bits exactement et E écrit sur 8 bits en $NAT_8$. On retient alors sE_7E_6E_5E_4E_3E_2E_1E_0M_1M_2M_3M_4...M_{23}.

> **Exceptions :** pour E=0,M=000\,0000\,0000\,0000\,0000\,0000, c'est un cas particulier : x=0. Pour E=255,M=000\,0000\,0000\,0000\,0000\,0000, on ne calcule pas x et on dit que x=\pm \infty (selon la valeur du signe). Une valeur extrêmement spéciale est aussi définie pour E=255,M\neq 000\,0000\,0000\,0000\,0000\,0000. C'est `NaN`, *not a number*, qui est renvoyée lors de calculs impossibles (racines carrées ou logarithmes de nombres négatifs, division 0/0).

Cette norme de codage s'apelle le codage IEEE 754.

Les limites du codage sont:

- Le nombre positif le plus proche de 0 vaut 2^{-126} (l'opposé en négatif).
- Le nombre le plus grand qui n'est pas +\infty est (2−2^{−23})\times 2^{127} soit environ 3\times 10^{38}

Pour les limites de la précision, une explication complète est disponible. Python utilise des flottants doubles précisions (avec non pas 23, mais 53 bits de mantisse) et des exposants plus grands. Mais il faut en particulier retenir que :

- Les fractions qui n'ont pas comme dénominateur une puissance de 2 ont une représentation infinie en binaire (comme \frac{1}{3} en décimal). Elles sont donc sujettes à des approximations.
- L'égalité de flottants est dangereuse : il vaut mieux tester la proximité avec une certaine précision.
- Pour les grandes valeurs, tous les entiers ne sont pas représentables ; pour les très petites valeurs a ajoutées à une plus grande b, le résultat peut être b.

In [9]:
```
u.activite("Précision des flottants: disparition des entiers")
a=2**54 # 2 puissance 54
for i in range(50,55):
    a=2**i
    b=float(a)
    c=b+1
    u.mark("La différence entre $2^{{ {0} }}+1$ et $2^{{ {0} }}+1={2}$ est ${1}$".format(i,c-b,a)) # Devrait être
```

### Activité : Précision des flottants: disparition des entiers

La différence entre 2^{ 50 }+1 et 2^{ 50 }+1=1125899906842624 est 1.0

La différence entre 2^{ 51 }+1 et 2^{ 51 }+1=2251799813685248 est 1.0

La différence entre 2^{ 52 }+1 et 2^{ 52 }+1=4503599627370496 est 1.0

La différence entre 2^{ 53 }+1 et 2^{ 53 }+1=9007199254740992 est 0.0

La différence entre 2^{ 54 }+1 et 2^{ 54 }+1=18014398509481984 est 0.0

In [10]:

### Activité : Précision des fractions

Activité : Précision des fractions

**Partons du nombre 1, divisons le puis multiplions le par un même nombre.**

Aucune erreur jusqu'à 10, c'est super !

En allant jusqu'à 100, je n'ai que 2.0% d'erreur, c'est (presque?) rien !

*49, 98*

En allant jusqu'à 1000, je n'ai que 8.2% d'erreur, c'est (presque?) rien !

*49, 98, 103, 107, 161, 187, 196, 197, 206, 214, 237, 239, 249, 253, 322, 347, 374, 389, 392, 394, 412, 417, 425, 428, 443, 474, 478, 479, 491, 498, 499, 501, 503, 506, 509, 561, 569, 644, 685, 691, 694, 725, 729, 735, 737, 748, 753, 765, 778, 779, 784, 788, 789, 797, 809, 817, 823, 824, 829, 833, 834, 837, 841, 849, 850, 853, 856, 857, 886, 895, 927, 929, 941, 947, 948, 949, 956, 958, 969, 982, 996, 998*

En allant jusqu'à 10000, je n'ai que 11.8% d'erreur, c'est (presque?) rien !

En allant jusqu'à 100000, je n'ai que 13.1% d'erreur, c'est (presque?) rien !

...finalement, est-ce bien raisonnable ?

**Mais en fait, ce n'est pas très différent !**

Différence maximale pour \frac{1}{n}\times n pour n allant de 1 à 10 = 0

Différence maximale pour \frac{1}{n}\times n pour n allant de 1 à 100 = 4.440892098500626e-16

Différence maximale pour \frac{1}{n}\times n pour n allant de 1 à 1000 = 1.887379141862766e-15

Différence maximale pour \frac{1}{n}\times n pour n allant de 1 à 10000 = 5.995204332975845e-15

Différence maximale pour \frac{1}{n}\times n pour n allant de 1 à 100000 = 1.1657341758564144e-14

---

```
Mais en fait il suffit de regarder 0.2 !
0.2 = 0.2
Avec plus de précision (23 chiffres après la virgule) :
0.20000000000000001110223
```

---

Du coup certaines formules sont fausses : (a+b)(a-b)=a^2-b^2 pour a=0.0009 et b=0.9995

(0.0009+0.9995)(0.0009-0.9995)-(0.0009^2-0.9995^2)=-2.220446049250313e-16

In [11]:

Activité : Conversion réels ⟷ IEEE754

Que vaut 19,5 en IEEE 754 affiché en hexadécimal ? > _____

Que vaut -7,5 en IEEE 754 affiché en hexadécimal ? > _____

Que vaut -46,25 en IEEE 754 affiché en hexadécimal ? > _____

Que vaut 0,312~5 en IEEE 754 affiché en hexadécimal ? > _____

Que vaut 0,312~5 en IEEE 754 affiché en hexadécimal ? > _____

Que vaut +\infty en IEEE 754 affiché en hexadécimal ? > _____

Que vaut -26,375×2^{40} en IEEE 754 affiché en hexadécimal ? > _____

# Opérations dans les systèmes positionnels

## Addition

L'addition dans tous les systèmes positionnels se fait de la même façon, en décimal comme en binaire.

L'addition en base B se fait de la droite vers la gauche, colonne par colonne, en utilisant le fait que la somme de chiffres s'écrit sous la forme s + Br, où s est la somme partielle (un chiffre unique) et r est la retenue. Le poids de r est B fois plus important, et r est donc remise dans la colonne d'à côté.

Toutefois, il faut remarquer que les retenues sont à calculer dans la base utilisée: si on additionne des nombres binaires, il suffit d'attendre 2 (qui s'écrit 10 en binaire) pour avoir une retenue, et non pas 10 (en décimal).

Exemple:

| Poids | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|
| Retenues | ① | ① | ⑩ | ① | . | . |
| . | . | . | . | 1 | 0 | 0 |
| + | . | 1 | 0 | 1 | 1 | 0 |
| + | . | . | 1 | 1 | 1 | 1 |
| — | — | — | — | — | — | — |
| . | 1 | 0 | 1 | 0 | 0 | 1 |

> **NB :** On peut aussi marquer la retenue 10 avec 0 dans la colonne suivante et 1 dans la colonne d'ordre encore supérieur.
>
> **Astuce :** pour additionner une colonne, on peut bien sûr le faire en décimal, à condition de repasser au binaire pour le reste et les retenues.

## Multiplication

Méthode (Multiplication) À l'instar de l'addition, la multiplication se fait comme pour les nombres décimaux. Les tables sont justes différentes (il faut les écrire dans la bonne base !). En binaire, c'est très facile : on multiplie par 0 ou par 1, donc on se contente d'additionner des copies du multiplicande décalées là où le multiplicateur a des 1.

> **Remarque :** Très important : décaler à gauche de n colonnes un nombre, c'est le multiplier par B^n. Le décaler à droite, c'est le diviser par B^n.

Exemple:

| Poids | 32 | 16 | 8 | 4 | 2 | 1 |
|---|---|---|---|---|---|---|
| . | . | . | . | 1 | 1 | 0 |
| × | . | . | . | 1 | 0 | 1 |
| — | — | — | — | — | — | — |
| . | . | . | . | 1 | 1 | 0 |
| + | . | . | . | . | 0 | . |
| + | . | 1 | 1 | 0 | . | . |
| — | — | — | — | — | — | — |
| . | 0 | 1 | 1 | 1 | 1 | 0 |

## Autres opérations

Par analogie avec les techniques maîtrisées en base 10, il est possible de faire également :

- Des soustractions : lorsque le chiffre duquel on soustrait n'est pas suffisant, on ajoute 1 au chiffre à soustraire dans la colonne d'ordre immédiatement supérieur, et en compensation, on ajoute la base dans la colonne courante.
- Des divisions : en fait, on fait plein de multiplications enchaînées avec des soustractions.
- Additionner un négatif, c'est soustraire un positif.
- Des opérations en virgule fixe : les règles de placement de la virgule sont les mêmes qu'en base 10.
- Des décalages qui sont des multiplications ou divisions par une puissance de la base.
- Pour la virgule flottante, les multiplications sont simples ; les additions nécessitent de recoder en virgule fixe.

In [ ]:

## Relation avec les codages

Comme expliqué précédemment, nous allons agir sur une représentation différente avec les codages. Toutefois, les algorithmes vont être très semblables à ceux utilisés sur l'écriture positionnelle.

Si l'algorithme fait ce qu'il est censé faire, on dit qu'il est correct : il agit alors sur les représentations des concepts de la même façon que la fonction (d'addition par exemple) est censé le faire.

Mais parce que le codage a un domaine plus restreint que l'ensemble de tous les nombres, il est possible que le résultat n'appartienne pas aux nombres que l'on peut coder. On dit alors que le résultat n'est pas représentable.

Pour l'addition des naturels, par exemple, parce que l'on a juste complété par des 0 à gauche, l'algorithme d'addition classique a la bonne propriété : si le résultat d'une addition est représentable, alors cet algorithme est correct.

> **Théorème :** L'addition classique opérant sur les nombres codés en C2 vérifie la propriété que si le résultat est représentable, alors l'addition est correcte.

C'est un peu plus surprenant pour des C2, car la représentation des négatifs se fait par des nombres qui (en naturel) auraient des valeurs très élevées. Mais... ça marche (et ce n'est pas le magie : en fait le calcul est exactement l'addition dans $\raise{0pt}{\mathbb{Z}}\raise{-1pt}{/}\raise{-2pt}{n\mathbb{Z}}$, c'est-à-dire les nombres modulo $n$).

In [12]:

### Activité : Addition de codages

Faire les opérations suivantes en transformant les nombres au préalable en codage C2 sur 8 bits (résultat aussi en C2 sur 8 bits) :

Dites aussi si l'opération obtenue est correcte et représentable (réponse sans le 0b).

45+17 > _____

45-17 > _____

-17-17 > _____

17-45 > _____

221+45 > _____

## Opérateurs booléens

Il est possible d'interpréter les deux valeurs binaires comme représentant respectivement *vrai* (1) ou *faux* (0). C'est la logique booléenne. On peut définir des opérations correspondantes à la conjonction (et), la disjonction (ou) et la négation (opposé de).

Ce ne sont pas des opérations arithmétiques classiques, mais elles sont très utilisées.

| Opération | Anglais | Arithmétique | Logique | C bit-à-bit | C logique | Python |
|---|---|---|---|---|---|---|
| Conjonction (et) | AND | $c=ab$ | $c=a\wedge b$ | & | && | and |
| Disjonction (ou) | OR | $c=a+b$ | $c=a\vee b$ | \| | \|\| | or |
| Négation (non) | NOT | $b=\overline{a}$ | $b=\neg a$ | ~ | ! | not |
| Disjonction exclusive (ou bien) | XOR | $c=a\oplus b$ | $c=a\oplus b$ | ^ | != | != |

Les conventions du C sont reprises dans beaucoup d'autres langages : C++, Java, Javascript, par exemple. La différence entre les opérateurs bit-à-bit et logiques sera expliquée plus bas. Le XOR n'est pas aussi facilement accessible que les autres dans les langages, mais il peut souvent être remplacé par le symbole `!=` en transformant un peu les formules.

> **Exemple :** Soit $\mathcal A(p)$ la propriété « $p$ est un chemin existant » et $\mathcal B(p)$ la propriété « $p$ est un chemin qui désigne un répertoire ». Si on suppose qu'il n'y a que deux type d'objets (répertoires et fichiers), la propriété $\mathcal C(p)$ « $p$ est un chemin qui désigne un fichier » s'exprime par $\mathcal A(p)\overline{\mathcal B(p)}$.

### Les définitions

Les opérateurs sont définis par leur table de vérité.

| a | b | a AND b | a OR b | a XOR b |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

| NOT a |
|---|

| | a |
|---|---|
| 0 | 1 |
| 1 | 0 |

On retrouve quelques équations simples : (a+b)c=ac+bc

Mais d'autres qui ne sont vraies qu'en logique booléenne et pas en arithmétique générale :

a=a+ab

## Opérateurs bit à bit

Les opérateurs bit à bit sont des généralisations des opérateurs ci-dessus qui portent à chaque fois sur un seul bit (NOT) ou une paire (AND, OR, XOR).

Ces opérateurs agissent sur des chaînes de bits de même longueur, et appliquent l'opération indépendamment sur chaque bit.

> **Exemple :** NOT(011001) est la chaîne 100110. 1100\,0111 AND 0001\,1111 donne comme résultat 0000\,0111.

Les opérateurs bit à bit sont à distinguer des opérateurs globaux. Dans les langages où les booléens sont définis (Python par exemple), les opérateurs globaux agissent sur les booléens simples. Pour les autres, on réduit les nombres à une valeur booléenne en considérant la question « x est-il différent de 0 ».

> **Exemple :** En C, la valeur \mathrm{0b1011}\&\mathrm{0b1101} est égale à \mathrm{0b1001} : c'est l'opérateur bit à bit. En revanche la valeur \mathrm{0b1011}\&\&\mathrm{0b1101} est égale à 1 (pour être précis, les deux termes sont différents de zéro, sont donc transformés en 1, puis ensuite on fait 1\&1. C'est aussi ce qui se passe en Python, Javascript et Java. En shell, le `&&` permet d'enchaîner deux commandes en ne faisant la deuxième que si la première est vraie, ce qui est similaire à ce qui est le cas dans les autres langages (avec `||`, la deuxième instruction n'est exécutée que si la première est « fausse »).

On range aussi dans les opérateurs bit à bit souvent les décalages à gauche et à droite (<\!< et >\!>) qui correspondent au décalage de bits (mais aussi à la multiplication et la division par 2^i).

Ainsi, en C ou en Python, on peut calculer facilement et exactement faire ces opérations.

In [13]:
```
a=220
print(a>>1) # a/2
print(a<<3) # a*8
print(a>>8) # a/256
```

```
110
1760
0
```

In [14]:

### Activité : Opérateurs logiques

Combien vaut 0b0b01110000 AND 0b11101011 ? > _____

Combien vaut 0b0b10010100 OR 0b00001011 ? > _____

Combien vaut 0b0b11010101 XOR 0b00110011 ? > _____

Combien vaut NOT 0b01011111 ? > _____

Combien vaut 0b0b00111101 AND 0b10101010 ? > _____

Combien vaut 0b0b11011000 OR 0b01100001 ? > _____

## Masquage

Le masquage est représentatif d'une technique qui consiste à représenter plusieurs valeurs indépendantes qui sur peu de bits en les rassemblant au sein d'un même mot-mémoire.

Par exemple, si une donnée peut avoir plusieurs permissions : être autorisé à la lecture, à l'écriture, à la destruction, au déplacement, cela fait 4 valeurs indépendantes. Parce que les ordinateurs accèdent à la mémoire par des adresses qui comptent des octets (et non pas des bits), ces 4 valeurs prennent normalement 4 octets en mémoire. Il est toutefois possible de toutes les mettre sur le même octet, par

exemple une valeur entière qui serait constitué de 1 pour le droit de lecture, 2 pour le droit d'écriture, 4 pour le droit de destruction et 8 pour le droit de déplacement. Cela donne une valeur entre 0 et 15, qui est stockée sur un seul octet. Gain de place : 75%.

Une série de données booléennes condensées de cette façon s'appelle un *champ de bits*.

Le problème est que pour accéder à la valeur par exemple du droit à l'écriture, on doit déterminer si le nombre est obtenu avec un 2 ou pas.

C'est là qu'intervient le *masquage*. Le 2 ne sera présent que si le bit de poids 2^1=2 est présent dans le nombre. En faisant un AND (bit à bit) avec le nombre 0b0010 (2 en décimal), le seul bit qui peut rester à 1 est le bit 2.

La formule exacte pour récupérer la valeur du i-ième bit (de poids 2^i) est : b_x=(B\&(1\mathop{<\!<}x))\mathop{>\!>}x La formule pour mettre à 1 la valeur du i-ième bit (de poids 2^i) est : B=B|(1\mathop{<\!<}x) La formule pour mettre à 0 la valeur du i-ième bit (de poids 2^i) est : B=B\&(\sim(1\mathop{<\!<}x))

In [15]:

## Activité : Formules de masquage

Expliquez les formules ci-dessus.

In [16]:

## Activité : Masquage

Dans un système, la fonction `keyEvent()` renvoie une valeur entière sur 16 bits (dont 5 ignorés) :

- Les 8 premiers bits correspondent au numéro de la touche sur le clavier (pour les touches ordinaires)
- Le 9e bit (de poids 2^8) correspond à la touche SHIFT (1 : pressée, 0 : pas pressée)
- Le 10e bit (de poids 2^9) correspond à la touche CONTROL (1 : pressée, 0 : pas pressée)
- Le 11e bit (de poids 2^{10}) correspond au fait d'appuyer sur une touche (1) ou de l'avoir juste relachée (0)

Dans la cellule suivante, définissez une fonction qui décrit la touche en écrivant un texte du genre : « Vous venez de lâcher la touche 27 en ayant SHIFT appuyé et CONTROL lâché »

In [17]:
```
def reponse(a):
    texte="**je n'ai pas encore tapé le programme**"
    return texte
```

In [18]:
```
num=random.randint(0,2047)
u.mark("La réponse pour {0} est :".format("0b"+bin(num).replace("0b","").rjust(11,"0")))
u.mark(solution(num))
u.mark("Votre programme répond :")
u.mark(reponse(num))
```

La réponse pour 0b10011011010 est :

Vous venez de presser la touche 218 en ayant SHIFT pas pressée et CONTROL pas pressée

Votre programme répond :

**je n'ai pas encore tapé le programme**

In [19]:
```
u.soltoggle()
```

```
Mode corrigé
```