# résumé

## Numération par position

Différentes bases (10, 2 et 16) mais un même système : *numération par position.*

| systeme | base | chiffres |
|---|---|---|
| Décimal | 10 | 0123456789 |
| Binaire | 2 | 01 |
| Héxadécimal | 16 | 0123456789ABCDEF |

Convertir en décimal :

- **Décimal** : $2019 = 2 \times 1000 + 1 \times 10 + 9 = 2 \times 10^3 + 1 \times 10^1 + 9 \times 10^0$
- **Binaire** : $0b1101 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0 = 8 + 4 + 1 = 13$
- **Hexadécimal** : $0xA5F = 0xA \times 16^2 + 0x5 \times 16^1 + 0xF \times 16^0 = 10 \times 256 + 5 \times 16 + 15 = 2655$

Les *entiers* ne changent pas, leur *représentation* varie : `103 = 0b1100111 = '0x67'`

**Remarques :**

- Dans la mémoire d'une machine : binaire. À l'écran, ça varie, on peut avoir les trois.
- **Taille** (= nombre de chiffre) des résultats opérations. Même principe dans toutes les bases :
  - la taille d'une **somme** est $\leq$ à 1 + la plus grande des deux tailles.
  - la taille d'un **produit** est $\leq$ à la somme des deux tailles.

## Convertir en binaire, convertir en hexadécimal

On utilise l'algorithme des divisions successives :

1. On divise par 2 **jusqu'à ce que le quotient soit 0**
2. On lit les bits en montant de droite à gauche : `167 = 0b10100111`

Cet algorithme se programme en python :

```python
def bina(entier):
    if entier == 0:
        return "0"
    bits = ""
    while entier != 0:
        reste = entier % 2
        entier = entier // 2
        bits = str(reste) + bits
    return bits
```

La conversion `décimal -> hexadécimal` se fait par des divisions par 16.

divisions successives

Figure 1: divisions successives

demi-additionneur

Figure 2: demi-additionneur

## Information dans la machine.

On regroupe les bits par paquet de 8 bits : 8 bits = 1 octet.

Attention, en anglais : octet se dit *byte* !

Les notations : 1000 bits = 1 kilo bit = 1kb ; 1000 octets = 1 kilo octet = 1ko = 1kB

Il faut 2 chiffres hexadécimaux pour représenter un octet : `0xaf = 175`

## Booléens et portes logiques.

George Boole (1815-1864) créateur de l'*algèbre de Boole* qu'on utilise pour représenter la logique interne de la machine.

Utilise deux états "Vrai : 1", "Faux : 0". Notés `True` et `False` en python.

### Opérateurs booléens.

| opérateur | description | exemple | python |
|---|---|---|---|
| `non` | contraire | `non vrai = faux` | `not` |
| `et` | et logique | `vrai et faux = faux` | `and` |
| `ou` | ou logique | `vrai ou faux = vrai` | `or` |
| `xor` | ou exclusif | `vrai xor faux = vrai` | pas implémenté en Python |

On utilise des *tables de vérité* pour les représenter :

| A | B | A XOR B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

### Circuits : le demi additionneur

La porte logique du haut est un XOR et celle du bas un AND.

Il est appelé *demi-additionneur* car il réalise l'addition de 2 bits (**A** et **B**), le résultats de cette somme est représentée par **S** et la retenue éventuelle par **R**.

## Opérations bits à bits

Une opération "bits à bits" est réalisée pour chaque bit des deux entrées. Le nombre final est la combinaison des résultats.

Exemple : un **masque** est un "`et` bit à bit".

| | |
|---|---|
| nombre | 1011.1100 |
| masque | 1101.0101 |
| et bit à bit | 1001.0100 |

## Représentation d'un texte en machine

> Attendu : Identifier l'intérêt des différents systèmes d'encodage. Convertir un fichier texte dans différents formats d'encodage.
>
> Commentaires : Aucune connaissance précise des normes d'encodage n'est exigible.

La machine ne "comprend" pas les textes. Elle attribue à chaque symbole un entier dans une table prédéfinie. C'est l'**encodage** de ce caractère. Il existe de nombreux encodages.

**Pourquoi différents encodages de caractères ?**

Parce qu'ils sont tous imparfaits. L'encodage a longtemps été *local* : chaque pays avait le sien...

## ASCII

ASCII (***American*** *Standard Code for Information Interchange*) est la première norme largement utilisée pour encoder des caractères. Comme son nom l'indique cette norme est américaine et elle n'inclut donc que les **lettres latines non accentuées**, les chiffres, des symboles de ponctuation et certains caractères spéciaux.

Voici les caractères de la table ASCII (les 33 premiers, et le dernier, ne sont pas imprimables) :

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | NUL | SOH | STX | ETX | EOT | ENQ | ACK | BEL | BS | HT | LF | VT | FF | CR | SO | SI |
| 1 | DLE | DC1 | DC2 | DC3 | DC4 | NAK | SYN | ETB | CAN | EM | SUB | ESC | FS | GS | RS | US |
| 2 | ESP | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / |
| 3 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 4 | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5 | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 6 | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 7 | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ~ | DEL |

128 caractères composent la table ASCII, ce qui permet de les représenter sur 7 bits (en pratique plutôt 8 bits afin d'occuper un octet complet).

## ISO-8859-1

Par la suite d'autres encodages ont vu le jour afin de pallier les limites de l'ASCII. L'ISO-8859-1 (aussi appelé *Latin-1*), pour l'Europe occidentale, a vu le jour en 1986. Celui-ci comble les manques pour la plupart des langues d'Europe occidentale. Pour le français il manque cependant le œ, le Œ et le Ÿ et, bien entendu, le symbole €. L'encodage en ISO-8859-1 utilise 8 bits, les 128 premières valeurs de l'ISO-8859-1 sont identiques à l'ASCII, ce qui assure une compatibilité avec cet encodage.

C'est l'encodage courant sous windows.

## UTF-8

**Cet encodage est le meilleur à ce jour. On l'utilise autant que possible**

À nouveau le codage ISO-8859-1 (et les autres codages de la famille ISO-8859) présentent des limites. Dans les années 1990, le projet Unicode de codage unifié de tous les alphabets est né. Différents codages sont utilisés pour représenter des caractères Unicode (UTF-8, UTF-16, UTF-32...). Ici nous nous concentrons sur l'UTF-8

Le codage UTF-8 est un codage de longueur variable. Certains caractères sont codés sur un seul octet, ce sont les 128 caractères du codage ASCII. Les autres caractères peuvent être codés sur 2, 3 ou 4 octets. Ainsi l'UTF-8 permet en théorie de représenter $2^{21} = 2\,097\,152$ caractères différents, en réalité un peu moins.

Actuellement ~100.000 caractères encodés en UTF-8.

Les caractères en UTF-8 doivent avoir une forme particulière décrite dans la table ci-dessous :

| Nbre octets codant | Format de la représentation binaire |
|---|---|
| 1 | 0xxxxxxx |
| 2 | 110xxxxx 10xxxxxx |
| 3 | 1110xxxx 10xxxxxx 10xxxxxx |
| 4 | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx |

L'encodage UTF-8 est lui aussi compatible avec l'ASCII. En revanche ISO-8859-1 et UTF-8 sont incompatibles entre eux pouvant conduire à des caractères illisibles.

## Complément à deux : comment coder les entiers négatifs dans une machine ?

Ce qu'on a vu jusqu'ici ne permet que d'encoder des nombres positifs.

**Deux approches possibles :**

**Approche insatisfaisante : binaire signé**

On fixe la taille mémoire de chaque nombre (par exemple 4 bits). Le premier bit est le **bit de signe** : 1 pour les négatif, 0 pour les positifs.

- En binaire signé sur 4 bits : `0b1001 = -1` , `0b1011 = -3`, `0b0111 = 7` etc.
- L'addition présentée plus haut **ne fonctionne plus** : `0b1001 + 0b0001 = -1 + 1 = 0` mais bit à bit cela donne : `0b1010 = -2`... Le binaire signé c'est nul.

**Complément à deux**

on fixe la taille mémoire de chaque nombre (par exemple 4 bits).

- Les nombres positifs sont encodés comme d'habitude.
- Les nombres négatifs sont encodés ainsi : (exemple : -3)
  1. Coder la valeur absolue du nombre en base 2 : `3 = 0b11`
  2. compléter l'octet avec des 0 devant (jusqu'à la taille): `0b0011`
  3. échanger tous les bits ($1 \leftrightarrow 0$) : `0b1100`
  4. ajouter 1. `-3 = 0b1101`
- Le complément à deux permet de conserver le même algorithme pour l'addition :

  `-3 + 2 = 0b1101 + 0b0010 = 0b1111 = -1`
- le complément à deux c'est bien... MAIS il faut prédéfinir une taille !!!

  Pas d'entiers de taille *arbitraire* en complément à deux !

`1111 0011 = ???`

Devant une série de bits, on ne peut **deviner** ce qu'ils représentent.

Est-ce un entier positif ? négatif (binaire signé, complément à deux) ? autre chose ?

Le contexte (pour vous : l'énoncé) précise ce qu'il faut comprendre.

**Table de valeurs du complément à 2 sur 8 bits**

```
 bit
 de
signe
0 1 1 1 1 1 1 1 =  127
    0       ...      =  ...
    0 0 0 0 0 0 1 0 =    2
    0 0 0 0 0 0 0 1 =    1
    0 0 0 0 0 0 0 0 =    0
    1 1 1 1 1 1 1 1 =   -1
    1 1 1 1 1 1 1 0 =   -2
    1       ...      =  ...
    1 0 0 0 0 0 0 1 = -127
    1 0 0 0 0 0 0 0 = -128
```

**et Python là dedans ?**

Les opérations précédentes ont toutes supposées une taille fixe des entiers : **codés sur un octet** par exemple.

Dans Python les entiers ont une *taille arbitraire*, il ne peut afficher nativement le complément à deux.

```
>>> bin(12)
'0b1100'
>>> bin(-12)
'-0b1100'
```

## Flottants

Afin de représenter des nombres à virgule en machine plusieurs approches sont envisageables : la *virgule fixe* ou la *virgule flottante*.

Pour l'un comme l'autre l'espace mémoire est le même : chaque nombre occupe le même nombre de bits. La précision possible est très différente.

### Virgule fixe

On décide une fois pour toute de l'emplacement de la virgule. Pour le reste c'est comme en binaire.

Par exemple, sur 8 bits, avec une virgule après 4 bits : `1011,0011`.

La précision est limitée par le nombre de bits après la virgule et la taille maximale aussi.

Peu commode.

### Virgule flottante

On s'inspire de la notation scientifique : $3.45678 \times 10^3 = 3456.78$.

Et donc, pour un flottant :

$1.1001101 \times 2^4 = 1\ 1001.101$ soit $16 + 8 + 1 + \frac{1}{2} + \frac{1}{8}$.

Les bits après la virgule sont les *inverses* des puissances de 2.

En machine, il faut encoder plusieurs informations:

- le signe,
- l'exposant,
- les bits *après* la virgule. *Pour le premier, c'est toujours 1, on peut l'ignorer.*

C'est ce qui est fait dans la norme IEE 754 de 1985.

### IEE 754

Dans cette norme (IEEE 754, double précision), les nombres dyadiques sont codés sur 64 bits en réservant :

- 1 bit pour le signe $S$,
- 11 bits pour l'exposant $E$,
- 52 bits pour la mantisse $M$.

La valeur du nombre est alors :

$$(-1)^S \times M \times 2^{E-1033}$$

Ce qu'on peut résumer ainsi :

| Norme | Encodage | Signe | Exposant | Mantisse | Valeur | Précision |
|---|---|---|---|---|---|---|
| **Double précision** | 64 bits | 1 bit | 11 bits | 52 bits | $(-1)^S \times M \times 2^{E-1033}$ | 53 bits |

#### Erreurs de calculs

De la même manière qu'avec la notation scientifique, des erreurs apparaissent nécessairement lorsqu'on calcule avec des flottants.

La plus frappante est certainement

```
>>> 0.1 + 0.2
0.30000000000000004
```

Elle provient d'une erreur de *conversion* en binaire de 0.1, 0.2 et 0.3 qui ne sont pas *dyadiques*. C'est-à-dire qu'ils n'admettent pas de représentation *finie* en binaire.

#### Egalité impossible entre les floats.

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

Il est impossible de tester l'égalité de deux flottants.

#### Autres erreurs.

D'autres existent, comme par exemple des problèmes d'amplitude :

Avec la *double précision*, on ne peut représenter $10^{40000}$ qui dépasse largement le plus grand nombre qu'on puisse représenter : $2^{1024} - 2^{971} \approx 1,80 \times 10^{308}$.

On évite ces problèmes en proposant deux valeurs supplémentaires : `NaN` et les infinis.

#### `NaN` et infinis

- Les infinis se rencontrent lorsqu'un calcul dépasse le plus grand nombre représentable. Ils sont *plus grands* que toute valeur.
- les `NaN`, pour *not a number* se rencontrent lorsqu'on n'est pas capable de représenter un nombre par manque de précision. Par exemple, soustraire des infinis ou les diviser entre eux.