NSI 1ère - Données

QK

Comment stocker des informations dans une machine ?

Une idée relativement récente

Nous allons aborder rapidement les procédés employés pour stocker et manipuler de l'information.

Utiliser une machine pour calculer est une idée remontant à l'antiquité. Utiliser une machine pour **stocker de l'information** remonte au XVIIIeme siècle.

Historique sommaire

En 1725 on voit l'apparition des cartes perforées : feuilles de papier rigides sur lesquelles sont disposés des trous qui symbolisent des données.

On stockait, par exemple, les plans de conception de tricot jacquard

Une accélération récente

On a rapidement distingué la mémoire vive (qui se vide quand le courant ne circule plus) de la mémoire morte (dont l'information persiste).

Au XXeme siècle on utilise

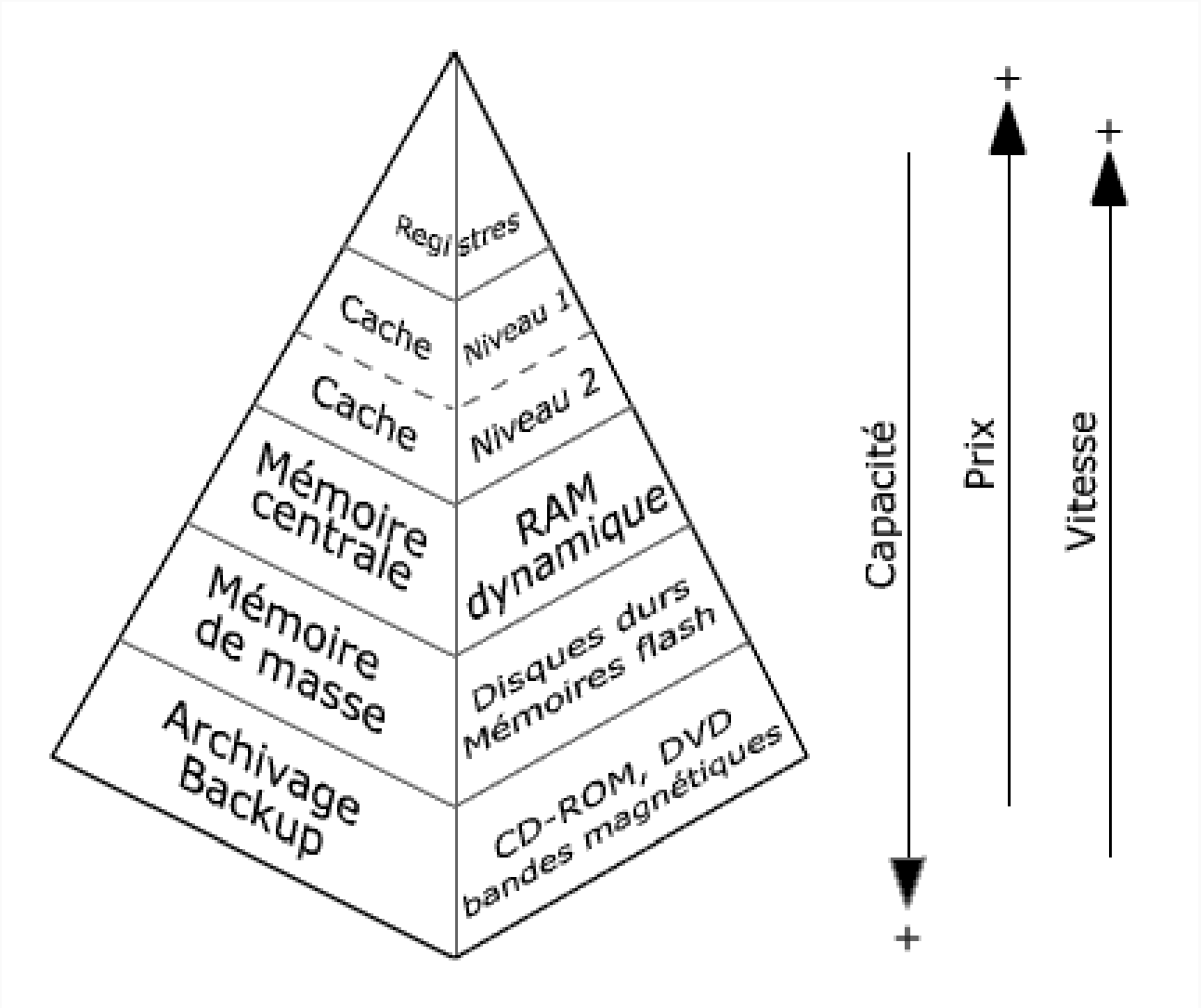
- l'électromagnétisme (aimants) et de la mécanique (ça tourne): bande, cassette, disquette et disques durs pour le stockage.
- de l'électronique pure pour la mémoire vive. 10^4 fois plus rapide…

Depuis une trentaine d’année la mémoire flash, plus rapide, sans élément mécanique, peu gourmande en énergie mais coûteuse et limitée en nombre de cycles de réinscription.

Pyramide de la mémoire

On peut résumer ainsi :

- Rapide = coûteux = limité en espace
- Lent = économique = vaste en espace


Registres
Cache
Niveau 1
Cache
Niveau 2
Mémoire centrale
RAM dynamique
Mémoire de masse
Disques durs
Mémoires flash
Archivage
Backup
CD-ROM, DVD
bandes magnétiques
Capacité
Prix
Vitesse
+
+
+
+

Données : deux unités de mesure

Nous avons 10 doigts et comptons avec 10 chiffres.

En informatique on emploie un autre système pour représenter les nombres :

les bits 0 et 1.

- 1 bit : 0 ou 1. Unité minimale de symbole b, parfois bit.
- On regroupe souvent les bits par paquet de 8 : 1 byte = 8 bits en anglais, symbole B ou 1 octet en français, symbole o

Données : ordre de grandeur

Préfixe	long	10^n	Exemple
kilo	milliers	10^3	$3,5$ kb = 3500 bits
mega	millions	10^6	1 Mb = 1 million de bits = 125 kB
giga	milliards	10^9	
téra	billions	10^{12}	1 TB = 8×10^{12} b
péta	billiards	10^{15}	

Confusions

Utiliser la même lettre crée des confusions : **b** ou **B**

Par exemple, les vitesses de transfert sont souvent exprimées en bits et le stockage en octet.

Quelques exemples

Objet	Espace mémoire
1 lettre	7 bits en ASCII
1 page de texte	3×10^4 bits
Disquette 3.5“	$1,44$ MB = $1,2 \times 10^7$ bits
Disque dur en 1980	20 MB = $1,6 \times 10^8$ bits
Bdd du WDCC	5000 TB = 4×10^{16} bits
Trafic internet (2016)	1.56×10^9 TB= $1,25 \times 10^{22}$ bits
1 gramme d’ADN	$1,8 \times 10^{22}$ bits

Nombres en informatique.

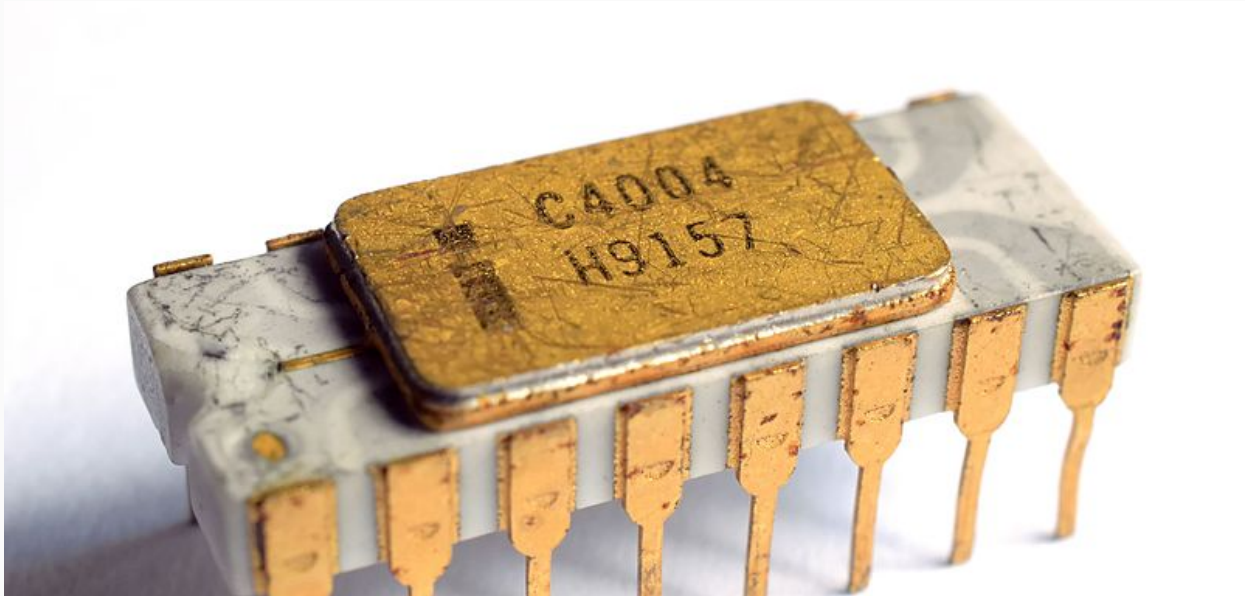
Pourquoi les bits de données ?

Partons de ce qu'on sait faire :

- On sait construire de très petits transistors.
- On sait les concentrer sur une petite surface.

Par exemple :

- 1971 : 2300 transistors dans un processeur 4004.
- 2014 : 2,6 milliards dans un core i7 d'intel.



Nos transistors permettent de contrôler un courant (ou une tension) sur un fil en sortie.

On peut enchaîner les transistors et réaliser des opérations logiques… en particulier l'addition !

En combinant tout ça, il est naturel d'opter pour une information stockée par paquets de bits. 1 pour "le courant passe", 0 pour le "le courant ne passe pas".

Les octets (paquets de 8 bits) furent choisis car 8 est la puissance de 2 la plus proche de 10.

On peut donc stocker $2^8 = 256$ bits dans un octet.

Les nombres en en informatique

On rencontre d'autres manières de représenter les nombres :

- binaire
- complément à 2
- octal
- hexadécimal etc.
- nombres à virgules flottantes

TP Binaire

Travail pratique

Nous représentons les valeurs entières dans le système décimal, on dit aussi en base 10.

- Dix chiffres de 0 à 9.
- La position des chiffres définit la valeur associée à ce chiffre.

$$542 = 5 \times 100 + 4 \times 10 + 2$$

Les différents chiffres correspondent aux puissances successives de 10 :

$$542 = 5 \times 10^2 + 4 \times 10^1 + 2 \times 10^0$$

L'information numérique, est représentée par des suites de 0 et de 1. Un **bit** peut prendre deux valeurs, 0 ou 1.

Manip :

D’où vient le mot bit ?

D’une base à l’autre

En binaire on écrit les valeurs entières en n’utilisant que les chiffres 0 et 1.

On utilise alors la base 2.

Les positions des chiffres sont associées aux puissances successives de 2

$2^0 = 1,\ 2^1 = 2,\ 2^2 = 4,\ 2^3 = 8,\ 2^4 = 16,\ 2^5 = 32$ etc.

Ainsi la valeur entière qui correspond à la représentation binaire 101010 est

$$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

$$= 1 \times 32 + 0 \times 16 + 1 \times 8 + 0 + \times 4 + 1 \times 2 + 0 \times 1 = 42$$

On distingue les notations en notant $0b101010$ ou 101010_2
Il nous faut pouvoir indiquer que 101010 est une représentation binaire et non une représentation décimale, qui serait comprise cent un mille dix. On notera 0b101010.

On distingue donc les valeurs entières (les entiers) et leur représentation.
Manip
Expliquez ce que peut signifier le signe '=' dans l'équation suivante
$10 = 2$ que l'on préférera écrire $0b10 = 2$

Changements de bases

Manip

- Donnez les valeurs entières représentées par $0b0100$, $0b10101, 0b101, 0b0101$ et $0b00101$.
- Comparez les valeurs entières représentées par $0b11$ et $0b100, 0b111$ et $0b1000$.

Manip

Quelle est la représentation binaire de 14 et 78 ?

Manip

De manière générale, quelle méthode employer pour trouver la représentation binaire d'une valeur entière ?

Binaire en Python

Python dérivant du langage C, les nombres en binaire sont notés `0bxxxx`

Python converti naturellement un entier **d'une base b vers le décimal** avec `int(nombre, b)`

La conversion **vers le binaire** se fait avec `bin` et renvoie une string

```
>>> a = '0b11'
>>> int(a, 2)
3
>>> b = 10
>>> bin(b)
'0b1010'
```

Pour l’instant nous n’avons parlé que d’entiers **positifs** et pour justifier le choix effectués concernant les entiers **négatifs**, il faut faire un peu de calcul.

TP : Calcul booléen

Opérateurs booléens

On définit sur ces valeurs booléennes trois opérations :

- la négation (le NON logique)
- la conjonction (le ET logique)
- la disjonction (le OU logique)

Le NON logique

Le NON logique d'un booléen a se définit par :

a	NON a
0	1
1	0

NON a vaut VRAI si et seulement si a vaut FAUX.

Le ET logique

Le ET logique entre deux booléens a et b se définit par :

a	b	a ET b
0	0	0
0	1	0
1	0	0
1	1	1

a ET b vaut VRAI si et seulement si a vaut VRAI et b vaut VRAI.

Le OU logique

Le OU logique entre deux booléens a et b se définit par :

a	b	a OU b
0	0	0
0	1	1
1	0	1
1	1	1

a OU b vaut VRAI si et seulement si a vaut VRAI ou b vaut VRAI.

Équivalences

Il est possible de définir l'opérateur OU logique à partir du NON logique et du ET logique.

En effet, si a et b sont des booléens alors a OU b = NON ((NON a) ET (NON b)).
On peut utiliser les tables de vérités pour démontrer cette égalité.

Preuve

On construit une table dans lesquelles les colonnes représentent les différentes sous-expressions dont nous avons besoin. Les contenus des colonnes sont construits en appliquant aux colonnes connues les tables de vérité connues définies ci-dessus.

Dans notre cas en plus de a, b, parmi les expressions utiles à notre calcul on trouve NON a, NON b. Une fois la table remplie pour ces deux expressions on peut déterminer celle de l'expression (NON a) ET (NON b) :

si on définit x=NON a et y= NON b,

alors (NON a) ET (NON b)=x ET y.

Table intermédiaire

a	b	NON a	NON b	(NON a) ET (NON b)
		x	y	x ET y
0	0	1	1	1
0	1	1	0	0
1	0	0	1	0
1	1	0	0	0

On complète alors la table avec les expressions : NON ((NON a) ET (NON b)) et (a OU b)

Table finale

a	b	(NON a) ET (NON b)	(NON a) ET (NON b)	a OU b
		(x ET y) = z	NON z	
0	0	1	0	0
0	1	0	1	1
1	0	0	1	1
1	1	0	1	1

L’égalité des contenus des deux dernières colonnes démontre l’équivalence des deux expressions.

a ET b

Manip

1. Trouvez une expression équivalente à a ET b construite uniquement à partir des opérateurs NON et OU.
2. Démontrez que votre proposition est correcte à l'aide des tables de vérité.

Lois de Morgan

Manip

1. Démontrez les règles de distributivité suivantes :
    1.1 a ET (b OU c) = (a ET b) OU (a ET c)
    1.2 a OU (b ET c) = (a OU b) ET (a OU c)
2. Démontrez les lois de Morgan :
    2.1 NON (a OU b) = (NON a) ET (NON b)
    2.2 NON (a ET b) = (NON a) OU (NON b)

OU exclusif

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

On rencontre également défini l'opérateur OU-exclusif, également appelé XOR (pour "eXclusive OR").

Manip

Démontrez l'équivalence :

a XOR b = (a ET (NON b))) OU ((NON a) ET b)

Les masques de sous-réseau

Très largement inspiré de cet article de Wikipedia.

Les adresses IP de version 4, IPv4, sont codés sur **32 bits**.

En notation décimale : 4 nombres compris entre 0 et 255, séparés par des points.

Par exemple : 192.168.100.2.

Elles sont ont composées de deux parties : le sous-réseau et l'hôte. Ils utilisent la même représentation.

On utilise des masques constitués (sous leur forme binaire) d'une suite de 1 suivis d'une suite de 0, il y a donc 32 masques réseau possibles.

Exemple de masque

Un exemple possible est le masque `255.255.255.0`.

Pour obtenir l'adresse du sous-réseau on applique l'opérateur ET entre les notations binaires de l'adresse IP et du masque de sous-réseau.

L'adresse de l'hôte à l'intérieur du sous-réseau est quant à elle obtenue en appliquant l'opérateur ET entre l'adresse IPv4 et la négation (NON) du masque.

Manip

1. Calculez le code binaire correspondant à l'adresse `192.168.100.2` (ou partez de l'adresse de votre machine).
2. Calculez le code binaire correspondant au masque `255.255.255.0`.
3. Calculez l'adresse binaire du sous-réseau puis donnez sa forme décimale.
4. Calculez l'adresse hôte puis donnez sa forme décimale.

A chaque porte est associée une représentation graphique. Voici pour les portes ET et XOR :

- porte ET

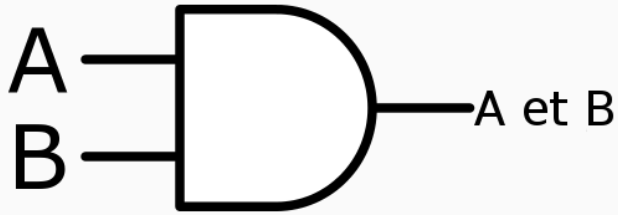


Figure 3: porte ET

- Porte XOR

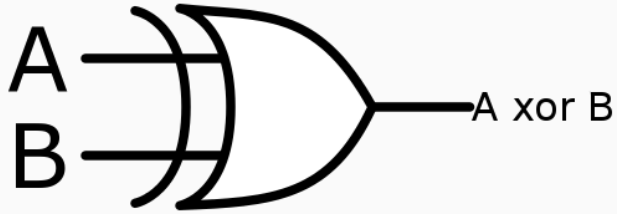


Figure 4: porte XOR

Portes logiques

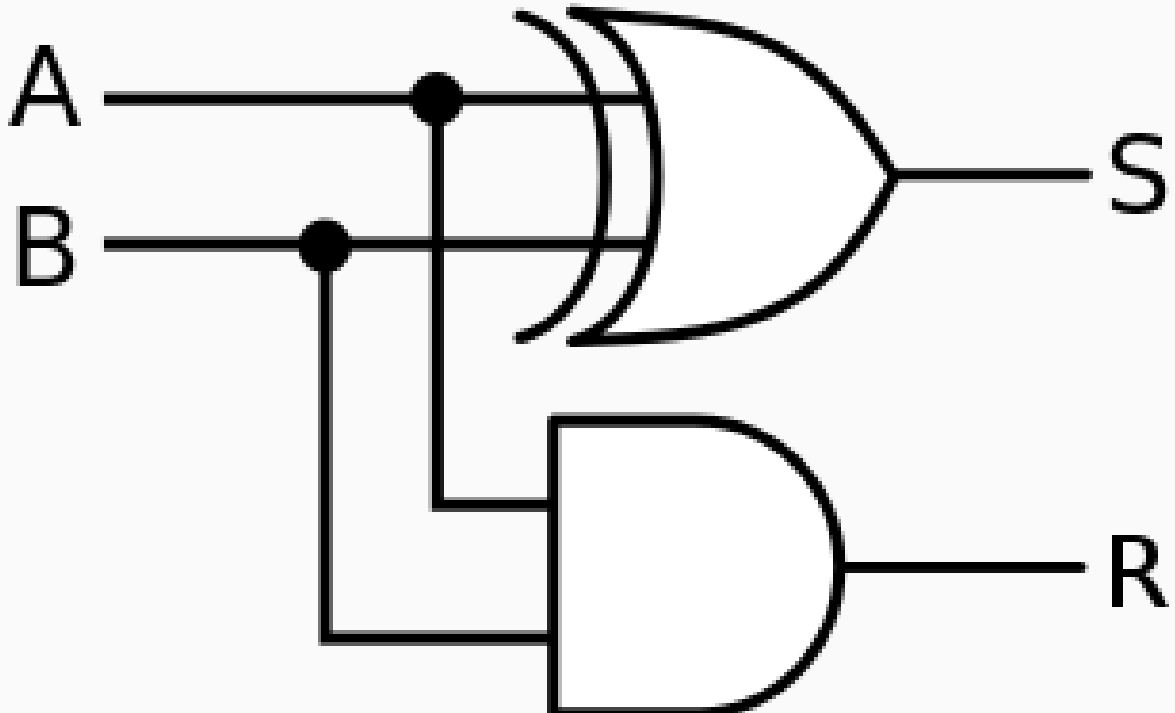
Les opérations logiques évoquées ci-dessus sont mises en oeuvre en électronique sous forme de **portes logiques**.

Les circuits électroniques calculent des fonctions logiques de l'algèbre de Boole.

Pour chacun des opérateurs logiques évoquées ci-dessus (et d'autres) il existe donc des portes logiques appelés porte ET, porte NON, etc. Les valeurs vrai et faux sont représentées par deux niveaux de tension, haut et bas.

Demi additionneur

Un circuit de type porte ET dispose donc de deux entrées et une sortie et la valeur du niveau de tension en sortie dépend des niveaux de tension appliquées à chaque entrée, en respectant la table de vérité du ET. Les portes peuvent être connectées entre elles pour réaliser des **circuits logiques** et on peut ainsi réaliser des calculs. Prenons l'exemple de ce circuit :



Étude du demi-additionneur

Manip

Vérifiez, avec une table de vérité, que **S** et **R** correspondent bien aux valeurs de la somme et de la retenue sur 1 bit de **A** et **B**.

Un exemple plus élaboré, le circuit 7400

Circuit intégré 7400 contenant 4 portes NON-ET (NAND). Les deux autres broches servent à l'alimentation 0V / 5V.

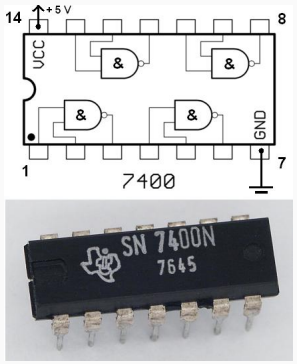


Figure 6: circuit intégré 7400

Combinaisons plus complexes

A partir de ce circuit on peut en construire d'autres plus complexes permettant d'additionner des nombres de plusieurs bits. Voir sur cette page par exemple.

Et on combine… jusqu'au micro-processeur qui réalise les calculs au sein d'un ordinateur. Il "suffit" de trouver la bonne organisation.

C'est un peu comme les Lego en somme… Vous pourrez trouver ici quelques compléments.

Nombres en binaires de plusieurs chiffres

Comment ajouter rapidement deux nombres en binaire ?

$5 + 4 = 9$ donc $0b101 + 0b100 = 0b1001$

On part du dernier bit (de poids faible) et on compte les retenues.

Il suffit donc de deux portes logiques pour réaliser une addition sur un bit : le calcul du bit se fait par un XOR et la retenue par un AND.

Il serait intéressant, pour limiter le nombre de composants de pouvoir décaler les bits. Ainsi, en décalant à droite et en conservant les retenues, on aurait toujours affaire au bit de poids faible.

Opérateurs en taille quelconque

On applique, bit par bit nos opérateurs usuels :

NOT bit à bit

Chaque bit est inversé.

Sur 4 bits, NOT 7 = 8

```
NOT 0111
  = 1000
```

ET bit à bit

Sur 4 bits, 5 AND 3 = 1 :

```
    0101
AND 0011
  = 0001
```

OU bit à bit

Sur 4 bits, 5 OR 3 = 7 :

```
   0101
OR 0011
 = 0111
```

XOR bit à bit

Sur 4 bits, 5 XOR 3 = 6 :

```
    0101
XOR 0011
  = 0110
```

Décalage de bits

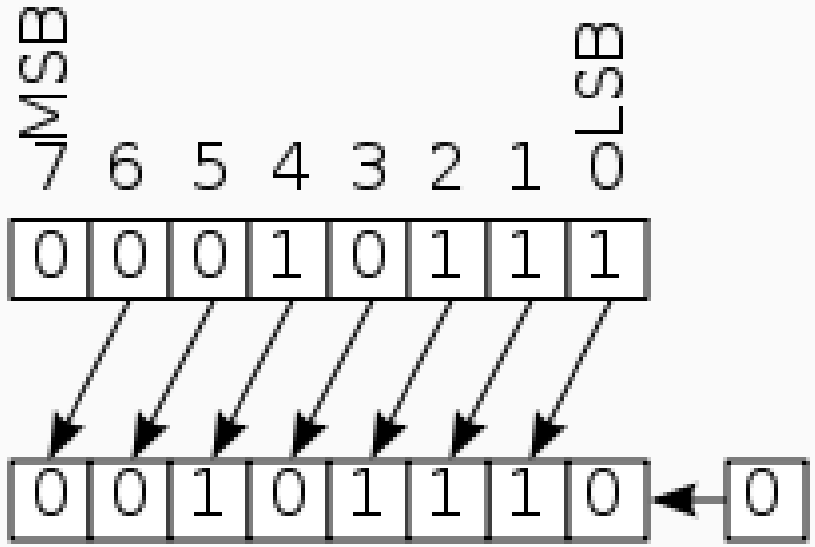
En décimal, un décalage à gauche est une multiplication par 10,

un décalage à droite, une division par 10 (entière)

En binaire :

- **décalage à gauche** : multiplication par 2
- **décalage à droite** : division par 2 (entière)

On supprime un bit d'un côté et on le remplace par un zéro.



Décalages à gauche et à droite Python

```
>>> # 1 * 2**5 = 32
>>> 1 << 5
32
>>> # 128 / 2**4 = 8
>>> 128 >> 4
8
```

Complément à 2

Mise en garde

Dans toute cette partie on travaille avec des binaires

SUR UN OCTET

L’idée de départ

- Une **représentation binaire des entiers négatifs** pour étendre **la même méthode d’addition à tous les entiers.**
- Si on ajoute un nombre avec celui obtenu en échangeant les 1 et les 0, on obtient 1111 1111.
- Ce n’est pas zéro mais presque…

Un exemple

6 s'écrit en binaire sur un octet : 0000 0110.

- On échange les 0 et les 1, on obtient : 1111 1001.
- On ajoute les deux nombres, on obtient : 1111 1111.
- On ajoute 1 à ce nombre, on obtient : (1) 0000 0000.
  En **négligeant le dépassement**, on obtient 0.
- L'opposé de 6 est donc 1111 1001 +1 = 1111 1010. Ce nombre s'appelle le complément à deux.

Une méthode plus rapide

On garde tous les 0 de droite jusqu'au premier 1, ainsi que ce 1. Pour le reste, on échange tous les 1 et tous les 0.

Exemples

- Le nombre 24 s'écrit : 0001 1000.
  Son opposé -24 s'écrit : 1110 1000.
- Le nombre 36 s'écrit : 0010 0100.
  Son opposé -36 s'écrit : 1101 1100.

Signe d'un entier dans le complément à 2

- Si le bit de poids fort est 1, l'entier est négatif. **1**110 1101 est négatif
- Si le bit de poids fort est 0, l'entier est positif. **0**010 1011 est positif

On peut représenter les entiers de $-128 = -2^7$ à $+127 = 2^7 - 1$ sur un octet

Table de valeurs

```
 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
```

Revenir au décimal depuis le complément à 2

- Si le bit de poids fort est 0, on fait comme d'habitude
- Si le bit de poids fort est 1, on change d'abord le signe. Sans oublier de rechanger à la fin !

Est-ce que ça remplit le contrat ?

- $36 + (-24) = 12$

```
  0010 0100
+ 1110 1000
= 0000 1100
```

C’est bien 12

- $24 + (-36) = -12$

```
  0001 1000
+ 1101 1100
= 1111 0100
```

Dont l’opposé est 0000 1100 soit 12

Complément à 2 : résumé

On a trouvé une méthode permettant d'ajouter des entiers (et donc de faire les opérations habituelles…) qui fonctionne aussi avec les entiers négatifs

et Python là dedans ?

Aie, c'est compliqué. Les opérations précédentes ont toutes supposées une taille fixe des entiers : **codés sur un octet**

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

```
>>> bin(12)
'0b1100'
>>> bin(-12)
'-0b1100'
>>> 2**7
128
```

Hexadécimal

Hexadécimal

Les nombres en binaires sont longs.

On utilise souvent la base 16 pour les manipuler plus facilement.

16 chiffres : 0 1 2 4 5 6 7 8 9 A B C D E F

Convertir un binaire en hexa est facile.

Chaque paquet de 4 bits donne un chiffre hexa :

$$1010\,0011\,1011\,1100_2 = A3BC_{16}$$

Notations

- Maths : $A3BC_{16}$,
- Langage C et dérivés (Python) : `'0xA3BC'`
- CSS : `'#A3BC'`

On peut utiliser une table

$0_{hex} = 0_{dec} = 0_{oct}$	0	0	0	0
$1_{hex} = 1_{dec} = 1_{oct}$	0	0	0	1
$2_{hex} = 2_{dec} = 2_{oct}$	0	0	1	0
$3_{hex} = 3_{dec} = 3_{oct}$	0	0	1	1
$4_{hex} = 4_{dec} = 4_{oct}$	0	1	0	0
$5_{hex} = 5_{dec} = 5_{oct}$	0	1	0	1
$6_{hex} = 6_{dec} = 6_{oct}$	0	1	1	0
$7_{hex} = 7_{dec} = 7_{oct}$	0	1	1	1
$8_{hex} = 8_{dec} = 10_{oct}$	1	0	0	0
$9_{hex} = 9_{dec} = 11_{oct}$	1	0	0	1
$A_{hex} = 10_{dec} = 12_{oct}$	1	0	1	0
$B_{hex} = 11_{dec} = 13_{oct}$	1	0	1	1
$C_{hex} = 12_{dec} = 14_{oct}$	1	1	0	0
$D_{hex} = 13_{dec} = 15_{oct}$	1	1	0	1
$E_{hex} = 14_{dec} = 16_{oct}$	1	1	1	0
$F_{hex} = 15_{dec} = 17_{oct}$	1	1	1	1

Figure 8: Table de conversion

De l'hexadécimal vers le décimal

Pour convertir $4D5_{16}$ de l'hexa. vers le décimal,

on commence par le dernier chiffre :

- 5×16^0 et on recule :
- 13×16^1 (D correspond au nombre 13)
- 4×16^2

$$4D5_{16} = 5 \times 16^0 + 13 \times 16^1 + 4 \times 16^2 = 1\,237_{10}$$

Du décimal vers l'hexadécimal

- Divisions entières successives par 16 jusqu'à trouver 0.

  Les **restes** donnent les chiffres dans l'ordre **inverse**

$$959 = 59 \times 16 + 15 \longrightarrow F$$

$$59 = 3 \times 16 + 11 \longrightarrow B$$

$$3 = 0 \times 16 + 3 \longrightarrow 3$$

$$959_{10} = 3BF_{16}$$

```
>>> int('3BF', 16)
959
>>> hex(145)
'0x3bf'
```

Les couleurs

En informatique on distingue

- les couleurs à l'écran : synthèse additive
- les couleurs imprimées : synthèse soustractive

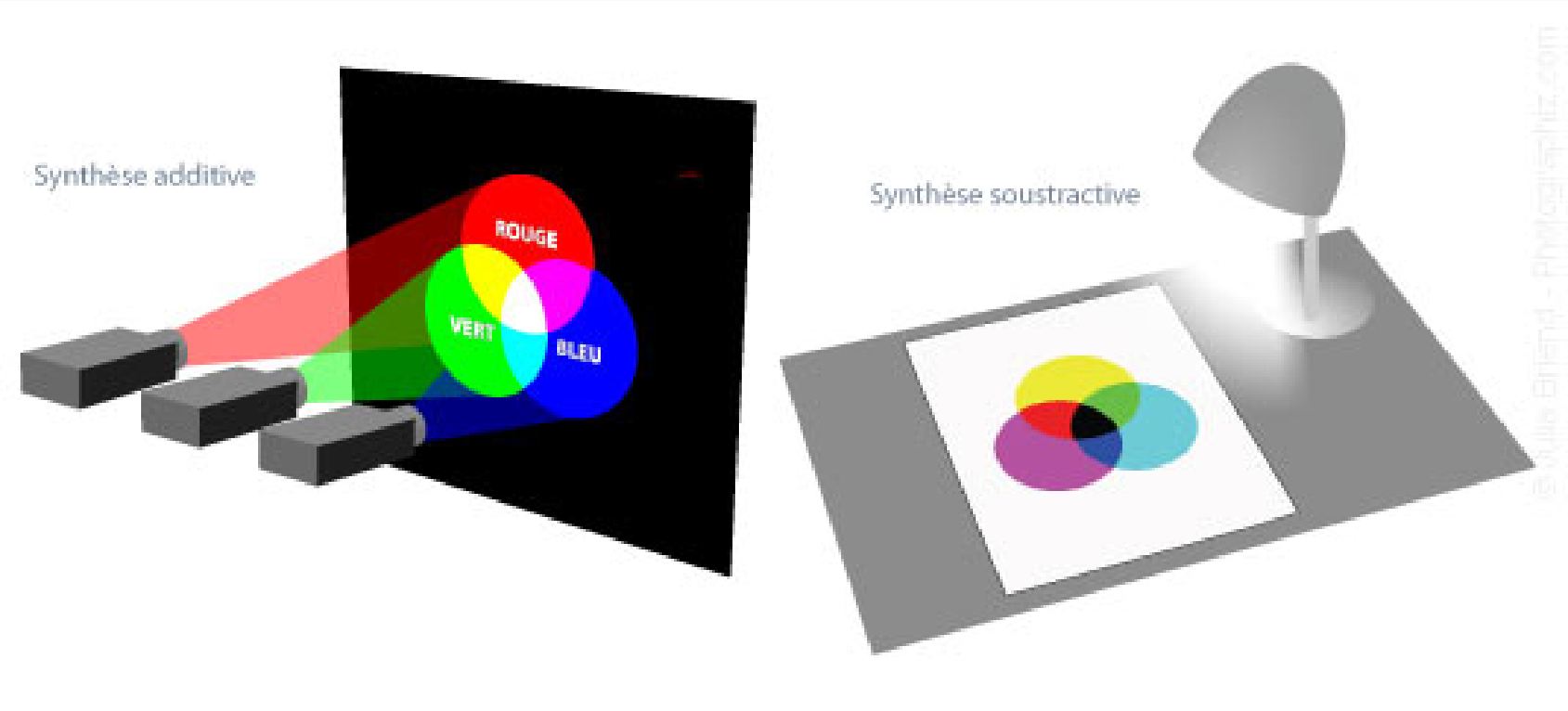
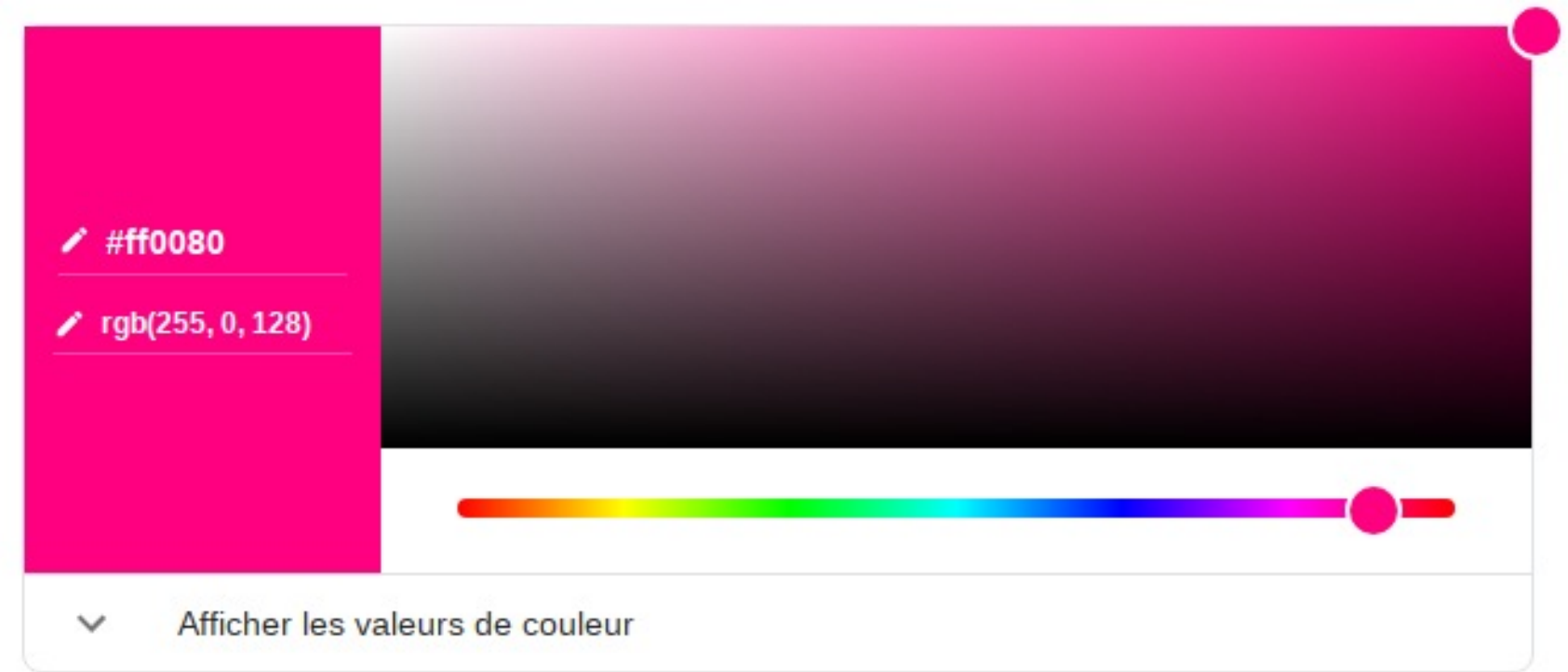


Figure 9: synthèses additive et soustractive

Synthèse additive

- En **synthèse additive** on utilise 256 niveaux de couleur pour les composantes Rouge, Vert et Bleu.
- Pratique de noter en hexadécimal : $256 = 16^2$
- #FF0080 : FF rouge à fond, 00 pas de vert, 80 bleu à moitié : un joli rose, noté parfois : rgb(255, 0, 128)



Quelques exemples

blanc	#FFFFFF	noir	#000000
rouge	#FF0000	jaune	#FFFF00
vert	#00FF00	cyan	#00FFFF
bleu	#0000FF	magenta	#FF00FF

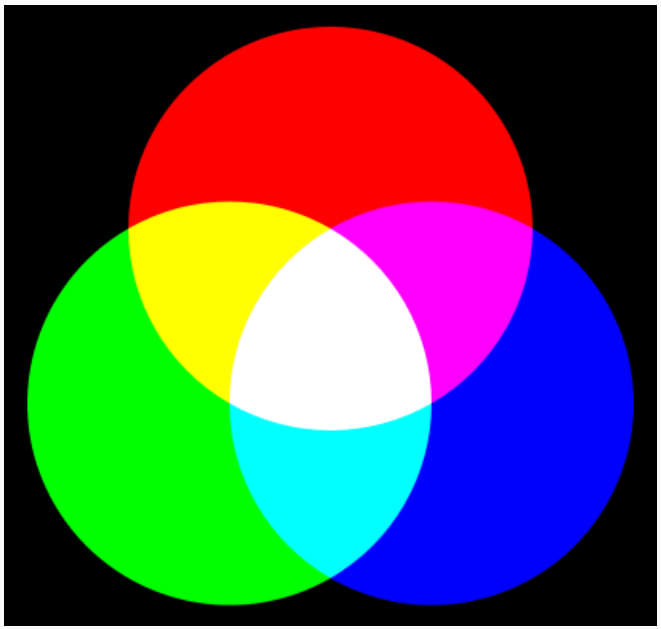


Figure 11: synthèse additive

Synthèse soustractive

- En **synthèse soustractive** on utilise souvent **CMJN** : cyan, magenta, jaune et noir.
- Le niveau de noir permet d'économiser les encres et améliore le rendu. On a développé de nombreuses méthodes.

Figure 12: synthèse soustractive

Nombre à virgule flottantes

0.30000000000000004

Partons d'un constat : Les ordinateurs savent manipuler les "nombres à virgules"

```
>>> 1.255465 * 753156.45
945561.5624992499
```

mais les résultats sont parfois surprenants :

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

Nombres à virgule flottante

Dans les machines, on utilise les **les nombres à virgule flottante**

Les nombres sont alors appelés des flottants (floats en anglais)

L'égalité de deux flottants n'a aucun sens

Décimaux

Nos machines travaillent en base 2 et les nombres à virgules flottantes sont représentés de la même manière.

Dans le système décimal on utilise les puissances de 10 avant et après la virgule : Par exemple 325,47 s'écrit

```
100      10     1.     1/10     1/100...
  3       2     5.     4        7
```

Nombres dyadiques

Dans la machine on utilise le même principe mais avec des puissances de 2.

On parle de nombres dyadiques

Par exemple : 7.625 = 4 + 2 + 1 + 1/2 + 1/8 et s'écrit en dyadique :

4	2	1.	1/2	1/4	1/8
1	1	0.	1	0	1

Revenons sur 0,1 + 0,2

0,1 et 0,2 ont des notiations décimales finies (ce sont des décimaux)

Leur notiation dyadique n'est pas finie !

$$0,1 = (0,0001100110011001100110011001100110011001100110011\cdots)_2$$

En machine elle est tronquée (mais sera très proche de 0,1)

Ce n'est pas génant en pratique : a-t-on souvent besoin d'une telle précision ?

Une approche naïve

Cette approche est intéressante et naïvement, on pourrait penser que la machine stocke ainsi ses nombres.

Problème : comment manipuler des nombres très grands et des nombres très petits en même temps ?

La taille de l'unvivers d'un côté, la taille d'un atome de l'autre !

- Il faudrait des milliers de chiffres…
- Les calculs sont compliqués…

Contourner la difficulté : la notation scientifique

Pour s'en convaincre :

$$A = 300000000 \times 0.00000015$$

Clairement la notation décimale n'est pas adaptée

On préfère la notation scientifique :

$$A = 300000000 \times 0.00000015$$

$$A = (3 \times 10^{8}) \times (1.5 \times 10^{-7})$$

Souvenons nous

- on multiplie 3 et 1,5 et
- on ajoute les puissances 8 et -7

$$A = (3 \times 1.5) \times 10^{8-7}$$

$$A = 4.5 \times 10^{1}$$

$$A = 45$$

Qu’est ce que ça donne en base 2 ?

Nombre à virgules flottante

Un **nombre dyadique** est représenté par :

$$\pm(1, b1 \cdots bk)_2 \times 2^e$$

où $b1, \ldots, bk$ sont des bits et e est un entier relatif.

Par exemple :

$$6,25 = (110,01)_2 = (1,1001)_2 \times 2^2$$

$$-0,375 = -(0,011)_2 = -(1,1)_2 \times 2^{-2}$$

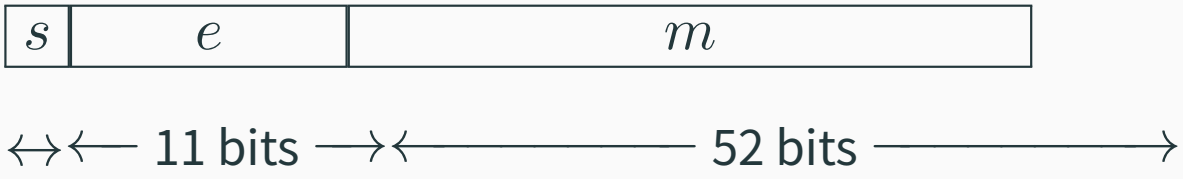
La suite de bits $b1 \ldots bk$ est la mantisse du nombre,

La puissance de 2 est l'exposant du nombre.

Stockage en mémoire

Dans cette norme, les nombres dyadiques sont codés sur 64 bits en réservant :

- 1 bit pour le signe ;
- 11 bits pour l'exposant ;
- 52 bits pour la mantisse.



Amplitude

Sans entrer dans les détails, en codant sur 64 bits on peut représenter des nombres entre :

- $2^{-1022} \approx 2,23 \times 10^{-308}$ pour le plus petit et
- $2^{1024} - 2^{971} \approx 1,80 \times 10^{308}$ pour le plus grand

Des améliorations sont faites pour les nombres très proches de 0.

Quand un flotant dépasse le plus grand nombre possible il est considéré comme infini

```
>>> 2.0 * 10**308 # dépasse le plus grand
inf
```

Quelques surprises avec `inf`

```
>>> a = float('inf') # rapide pour définir inf
>>> a
inf
>>> -a
-inf          # - inifini
>>> a + a
inf
>>> a - a    # opération interdite
nan           # not a number
>>> a + a == a
True
>>> b = 2.0 * 10 ** 309  # b = inf
>>> c = 2   * 10 ** 1000 # un integer
>>> c > b # inf est plus grand que tous les nombres
False
```

Deux problèmes dans les calculs avec les flottants

absorption

```
>>> (1. + 2.**53) - 2.**53     # = 1
0.0          # 1 a été absorbé par les gros nombres
>>> 1. + (2.**53) - (2.**53)   # on change l'ordre...
1                              # et ça fonctionne
```

Cancelation

Soustraire deux nombres proches fait perdre de la précision

Il peut y avoir des conséquences

- Le 25 février 1991, à Dharan en Arabie Saoudite, un missile Patriot américain a raté l'interception d'un missile Scud irakien, ce dernier provoquant la mort de 28 personnes. L'enquête a mis en évidence le défaut suivant :
- L'horloge interne du missile mesure le temps en 1/10s. Ce nombre n'est pas dyadique et est converti avec une erreur d'environ 0,000000095s
- Le missile a été mis en route 100h avant son lancement, ce qui entraine un décalage de

$$0,000000095 \times 100 \times 3600 \times 10 \approx 0,34s.$$

- C'est assez pour qu'il rate sa cible. Source

Représentation d’un texte en machine

Un caractère ?

Comment enregistrer, de manière optimale, du texte en mémoire ?

De combien de symboles a-t-on besoin ?

- 26 lettres dans l'alphabet, 52 avec les majuscules.
- 10 chiffres `0123456789`
- Un peu de ponctuation : `, ; : ! ? . / * $ - + = ( ) [] {} " '` etc.
- Quelques caractères techniques (retour à la ligne, espace etc.)

On dépasse $2^6 = 64$ mais en se contentant du minimum, on reste en dessous de $2^7 = 128$. On peut encoder une table assez vaste avec 7 bits.

Idée d'ASCII (1961) : uniformiser les nombreux encodages incompatibles entre eux.

La table ASCII complète

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Figure 13: La table ASCII

Remarques sur la table précédente

- Tout élément de la table est codé sur 7 bits, 1 octet par caractère suffit ($2^8 = 256$)
- Les chiffres commencent à 30_{16}, les majuscules à 41_{16} et les minuscules à 61_{16}
- Pour obtenir la notation binaire, on part de l'hexa.

  Premier chiffre : 3 bits, second chiffre 4 bits

$$A \to 41_{16} \to 4 \times 16 + 1 \to 0100\,0001$$

$$s \to 73_{16} \to 7 \times 16 + 3 \to 0111\,0011$$

- Seulement 95 caractères imprimables, pas de caractère accentués :

```
!"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~
```

Python et la table ascii

Les fonctions `chr` et `ord` permettent d'accéder à la table

```
>>> chr(65) # caractère 65 (décimal)
'A'
>>> ord('A') # numéro décimal du caractère
65
```

iso-8859-1 ou iso-Latin-1

Comment compléter la table ASCII ?

L'encodage iso-8859-1, dit iso-Latin-1 est apparu en 1986 et correspond à l'Europe de l'ouest. D'autres versions pour les caractères iso-Latin-2 de l'Europe de l'est etc.

- Reprend la table ascii et ajoute les accents au coût d'un octet supplémentaire.
- Encore incomplet : œ et Œ n'y sont pas !
  Ce qui a contribué à leur disparition de nombreux documents écrits dans les années 90…
- Windows (Windows-1252) et Mac (MacRoman) ont leurs versions
  Échange de documents et développement de logiciels **plus que pénibles.**

Bref, c'est ~~de la merde~~ imparfait.

Unicode

L'unicode et en particulier **UTF8** vise à résoudre TOUS les problèmes dans UNE norme.

- minimiser l'espace occupé par un caractère
- proposer un encodage adaptable à tous les caractères employés sur terre
- conserver l'ordre de la table ascii de départ

Unicode remonte à 1991, est encore en développement, comporte déjà 137 374 caractères d'une centaine d'écritures dont les idéogrammes, l'alphabet grec etc.

UTF8 est utilisé par 90,5% des sites web en 2017 et dans la majorité des systèmes UNIX (comprenez les serveurs)

Motivation d'unicode : $ et £

Les machines des années 1980 étant fournies avec leur propre encodage, une somme d'argent en dollars se voyait attribuer le symbole monétaire $ aux USA et le symbole £ au royaume uni (symbole monétaire de la livre sterling).

Mais 1$ $\neq$ 1£ et les confusions étaient fréquentes.

On a ensuite, peu à peu, étendu ce projet à tous les symboles existant.

Principe simplifié d'UTF8

- Chaque caractère est codé avec une séquence de 1 à 4 octets.
- Un texte encodé en ASCII est encodé de la même manière en UTF8 (sauf exception)
- Les premiers bits indiquent la taille de la séquence :
  - `0xxxxxxx` : 1 octet
  - `110xxxxx 10xxxxxxx` : 2 octets
  - `1110xxxx 10xxxxxx 10xxxxxx` : 3 octets
  - `11110xxx 1001xxxx 10xxxxxx 10xxxxxx` : 4 octets
- On note `U+XXXX` un caractère encodé en UTF8
- La taille est variable (génant pour les développeurs novices), l'espace en mémoire est parfois important
- Un caractère peut avoir plusieurs représentations $\rightarrow$ problèmes de sécurité informatique : certaines opérations interdites sont filtrées en reconnaissant des caractères. Ce problème est globalement résolu.

Python 2 et l'encodage des caractères

Ici prudence… **Python2** supporte bien UTF8 à condition de lui demander.

On trouve souvent dans l'entête d'un fichier .py :

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

qui signifient :

- Execute ce fichier avec `python`, situé dans le dossier `/usr/bin/env`
- **l'encodage du fichier est en utf8**

Sans quoi le premier accent va faire planter python.

Python 3 supporte nativement utf8, on peut se passer de cette précision

Python et l'UTF8

Une fois qu'on est assuré qu'UTF8 est supporté…

Alors c'est facile :

les fonctions chr et ord supportent unicode :)

```
>>> chr(29483)
猫 # qui signifie "chat" en chinois traditionnel
>>> ord(' 猫') # Et qui se prononce Māo
29483
```

TP Python

convertisseur de base calculer en binaire