NSI 1ère - Données

Le binaire et les opérations booléennes

qkzk

Les nombres en en informatique

Différentes bases

Dans la vie courante on utilise la base 10.

En informatique on rencontre d'autres manières de représenter les nombres :

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

Système de représentation par position

Qu'on utilise la base 10, 2, 8, 16 ou autre, on emploie toujours *la numération par position.*

La position des chiffres définit la valeur associée à ce chiffre.

numération par position

Exemple : En base 10, on a dix chiffres.

$345 = 3 \times 100 + 4 \times 10 + 5 = 3 \times 10^2 + 4 \times 10^1 + 5 \times 10^0$

En binaire, on a deux chiffres. Chaque chiffre est un *bit* (=**bi**nary digi**t**).

$0b1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13$

Du binaire au décimal.

Pour convertir un entier donné en binaire vers les décimal on le décompose.

Ensuite on effectue la somme. Cela va beaucoup plus vite en lisant de droite à gauche.

$0b10011 = 1 + 2 + 0 + 0 + 16 = 19.$

Autre notation $0b10011_2 = 19_{10}$

Du décimal au binaire.

Deux algorithmes majeurs.

- Facile *à programmer* : les divisions successives.
- Facile *de tête* : soustraire des puissances de 2.

Les puissances successives de 2 :

Exposant n	0	1	2	3	4	5	6	7	8	9	10	11
2^n	1	2	4	8	16	32	64	128	256	512	1024	204

Décimal au binaire avec les puissances de 2.

Écrire 57 en base 2 (=donner sa représentation binaire).

- $32 < 57 < 64$. Donc on fait $57 = 32 + 25 = 2^5 + 25$. On a un chiffre 1 à la position 6.
- $16 < 25 < 32$. Donc on fait $25 = 16 + 9 = 2^4 + 9$. On a un chiffre 1 à la position 5.
- $8 < 9 < 16$. Donc on fait $9 = 8 + 1 = 2^3 + 1$. On a un chiffre 1 à la position 4.
- $1 = 2^0$. On peut s'arrêter (dès qu'on atteint une puissance de 2). On a un chiffre 1 à la position 1.

$$57 = 0b111001$$

Autre exemple :

La représentation décimale de $0b101010$ est

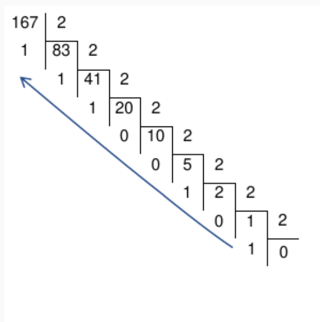
There Are 10 Types
Of People: Those
Who Understand
Binary And Those
Who Don't

Du décimal au binaire avec les divisions successives

Algorithme des divisions successives

Donner la représentation binaire de 167.

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`



Exercice

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

Exercice

Quelle est la représentation binaire de 14 et 78 ?

Exercice

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 *chaîne de caractères*.

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

Les nombres en mémoire

ATTENTION En mémoire, ce sont des entiers. POINT BARRE.

Ils sont encodés en binaire de toute manière.

Ainsi, pour réaliser l'opération 4+5 Python converti d'abord en binaire, additionne puis converti en décimal pour afficher 9.

Taille d'un nombre en binaire

Le **nombre de bit** d'un entier nous indique l'espace mémoire *minimal* qu'il faudra pour le stocker.

$123 = 0b1111011$ il faut au moins 7 bits pour stocker ce nombre.

En pratique, les machines utilisent des bloc de taille 1 octet, ce nombre entre dans un octet.

Taille et opérations

Si x occupe `n` bits et y occupe `p` bits alors :

- **SOMME** : $x + y$ occupe au plus `max(n, p) + 1` bits,
- **PRODUIT** : $x \times y$ occupe au plus `n + p` bits.

Nombre	x	y	$x + y$	$x \times y$
Nombre de bits	n	p	$\leq max(n, p) + 1$	$\leq n + p$

Calcul booléen

Booléen ?

Le terme *booléen* vient du nom du mathématicien britannique George Boole. Il est le créateur de la logique moderne qui s'appuie sur l'algèbre qui porte désormais son nom : l'*algèbre de Boole*.

Un **booléen** est une donnée dont la valeur ne peut prendre que deux états, soit l'état *vrai* soit à l'état *faux*. On utilise également le bit pour représenter des booléens : ainsi un 0 représente la valeur *faux* et un 1 représente la valeur *vrai*.

Un booléen en informatique

En python les booléens sont `True` et `False`.

Toute *comparaison* produit un booléen.

Par exemple, l'instruction `1==2` s'évalue à `False` :

```
>>> 1 == 2
False
```

Bien que le résultat soit faux, cette instruction est VALIDE.

Fonction qui retourne une booléen

On peut tout à fait retourner un booléen.

```
def est_majeur(age):
  return age >= 18
```

Et quand on l'exécute :

```
>>> est_majeur(22)
True
```

Affecter un booléen à une variable

```
>>> x = 9
>>> # imaginez ici 50 lignes de code
>>> comparaison = x == 3
>>> comparaison
False
```

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

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

Le OU logique

Le OU logique entre deux booléens a et b se définit par : a OU b vaut VRAI si et seulement si a vaut VRAI ou b vaut VRAI.

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

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.

Exercice

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

Exercice

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”).

Exercice

Démontrez l'équivalence :

$a \text{ XOR } b = (a \text{ ET } (\text{NON } b))) \text{ OU } ((\text{NON } a) \text{ ET } b)$

Opérateurs bits à bits en Python

```
x << y      # x décalé de y bits à gauche (vu plus tard)
x >> y      # x décalé de y bits à droite (vu plus tard)
x & y       # ET  bit à bit (ampersand)
x | y       # OU  bit à bit (tuyau)
~x          # NON bit à bit (tilde)
x ^ y       # XOR bit à bit (accent circonflexe)
```

Par exemple, pour le XOR bit à bit

```
0 ^ 0 = 0
0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 1 = 0
60        = 0b111100
30        = 0b011110
60 ^ 30   = 0b100010
0b100010 = 34
```

```
>>> 60 ^ 30
34
>>> bin(60 ^ 30)
'0b100010'
```

Et bit à bit

Exercice

1. Calculez la représentation binaire de 29.
2. Calculez la représentation binaire de 15.
3. Démontrer que le ET bit à bit entre 29 et 15 vaut 13.

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.

En fait, ce sont 4 *octets* généralement notés en décimal.

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.

Exercice

1. Calculez le code binaire correspondant à l'adresse `192.168.100.2`
2. Calculez le code binaire correspondant au masque `255.255.255.0`.
3. Vérifier que l'adresse binaire du sous-réseau est `192.168.100.0`
4. Vérifier que l'adresse de l'hôte est `0.0.0.2`

Conclusion : si le masque n'est constitué que de 255 et de 0 c'est facile.

Vers l'électronique et le calcul

Représentation graphique

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

- porte ET

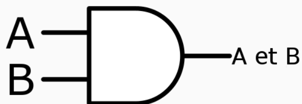


Figure 1: porte ET

- Porte XOR

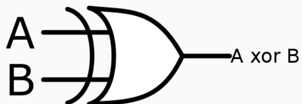


Figure 2: porte XOR

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.

La valeur du niveau de tension en sortie est obtenue avec 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.

Circuit du demi additionneur

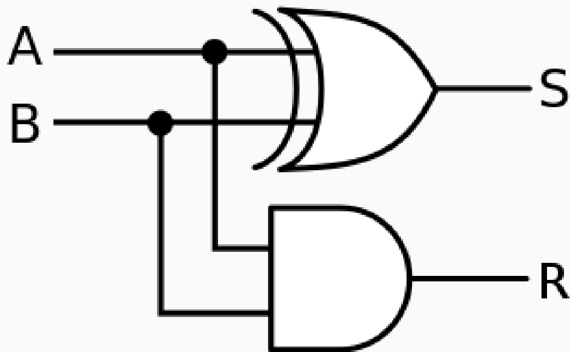


Figure 3: demi-additionneur

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**.

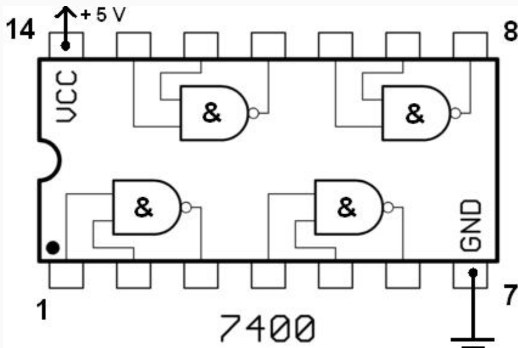
Étude du demi-additionneur

Exercice

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.



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
```