NSI en première (2019-2020)
Résumé Types simples

Quelques personnages importants

Le mathématicien Al Khwârizmî a écrit de nombreux ouvrages en arabe. La traduction de ces ouvrages en latin permet l'arrivée en Europe, seulement vers le XII^{e}, du zéro déjà utilisé par les mathématiciens arabes. Le zéro était déjà utilisé par les Indiens en tant que nombre à partir du V^{e}.

Gottfried Wilhelm Leibniz (1646-1716) construit une machine à calculer permettant d'effectuer des additions et des multiplications et expose les principes du calcul binaire.

George Boole (1815-1864) est considéré avec Augustus de Morgan comme le fondateur de la logique mathématique. (valeurs 1 pour vraie et 0 pour fausse).

Djon Atanasov (1903-1995) envisage la construction d'un calculateur électronique en utilisant le système binaire, en s'inspirant des idées de Leibniz. Toute information est codée en n'utilisant que des 0 et des 1. Avec Clifford Berry, il construit ce calculateur (sans programme enregistré). L'ABC (Atanasov Berry Computer) entre en service à la fin 1939. L'invention du premier ordinateur électronique est attribuée à Atanasov.

Presper Eckert et John Mauchly, conçoivent et construisent l'ENIAC dans les années 1940 sur la base des idées d'Atanasov.

1 Représentation numérique de l'information

L'utilisation uniquement de deux symboles facilite la mesure des états de circuits électroniques (ouvert ou fermé, faux ou vrai, 0 ou 1). Ces valeurs 0 et 1 s'appellent des chiffres binaires ou bit en anglais (pour binary digit). Une variable qui n'a que deux états comme 0 ou 1, ou alors faux ou vrai, s'appelle aussi un booléen (de George Boole).

Une machine reçoit de l'information, la mémorise, la modifie, la transmet à l'aide d'une multitude de petits circuits électroniques.

Avec 8 circuits, on peut construire un circuit qui décrit le mot 01000001 par exemple. Composé de 8 bits, on l'appelle un octet.

Remarque : avec 8 bits, on peut en particulier représenter l'ensemble des caractères disponibles sur un clavier.

Numérisation

Pour pouvoir être numérisée, l'information est d'abord discrétisée, puis représentée par une suite de 0 et de 1 et enfin enregistrée sur un support. Le type de support est unique et peut contenir toutes sortes d'informations en permettant de les voir ou de les entendre.

Un nombre, une touche du clavier (donc du texte) sont représentés par des nombres entiers, écrits ensuite en binaire.

Une image est décomposée en petits carrés nommés pixels (pour picture elements). La quantité de rouge, de vert et de bleu est aussi un nombre entier de 0 à 255 dans le système RGB par exemple.

2 Nombres entiers

2.1 Notion de base

La base dix

Dix chiffres sont nécessaires, (0, 1, 2, ..., 9). Le nombre dix s'écrit avec deux chiffres : 10.

Tableau avec les puissances de dix :

...	$1000 = 10^3$	$100 = 10^2$	$10 = 10^1$	$1 = 10^0$
...	7	2	5	3

L'utilisation du chiffre 0 est déterminante.

...	$10000 = 10^4$	$1000 = 10^3$	$100 = 10^2$	$10 = 10^1$	$1 = 10^0$
...			2	0	3
...	0	0	2	0	3

Quel que soit le nombre de 0 écrits à gauche, le nombre se lit "deux cent trois" et s'écrit 203.

Le mot zéro vient de l'arabe "sifr", traduction du mot indien "sunya" qui signifie "vide". Le mot arabe "sifr" est aussi à l'origine du mot "chiffre".

La base deux

En base deux il n'y a que deux chiffres, 0 et 1, et le nombre deux s'écrit avec deux chiffres 10.

On obtient l'écriture en base deux, ou l'écriture binaire, d'un nombre, avec les restes obtenus dans les divisions euclidiennes par 2 successives jusqu'à obtenir un quotient nul.

$11 = 5 \times 2 + 1, \quad r_1 = 1$
$5 = 2 \times 2 + 1, \quad r_2 = 1$
$2 = 1 \times 2 + 0, \quad r_3 = 0$
$1 = 0 \times 2 + 1, \quad r_4 = 1$

Nous obtenons l'écriture de 11 en base deux notée 1011_2, soit $r_4 r_3 r_2 r_1$.

Avec un tableau :

...	$8 = 2^3$	$4 = 2^2$	$2 = 2^1$	$1 = 2^0$
...	1	0	1	1

Un bit prend soit la valeur 0, soit la valeur 1. Un octet, byte en anglais, est constitué de 8 bits consécutifs.

L'octet est par exemple utilisé comme unité de référence pour mesurer la capacité des mémoires.

Remarque : pour multiplier par b un nombre écrit en base b, il suffit d'ajouter un zéro à droite du nombre.

Une base quelconque

Pour écrire les entiers naturels en base b, on a besoin de b chiffres et le nombre b s'écrit avec deux chiffres 10.

En base seize, on a besoin de 16 chiffres notés : 0, 1, 2, 3, 4, 5 , 6, 7, 8, 9, puis A (dix), B (onze), C (douze), D (treize), E (quatorze) et F (quinze).

Un octet s'écrit simplement en base seize. On partage l'octet en deux et chaque partie de quatre bits s'écrit avec un chiffre de la base seize.

Par exemple le nombre écrit 11010101 en base deux s'écrit D5 en base seize. En effet 11010101 se partage en 1101 et 0101 qui donnent respectivement D et 5 en base seize :

$11010101_2 = 1101_2 \times 10000_2 + 0101_2 = \mathrm{D}_{16} \times 10_{16} + 5_{16} = \mathrm{D5}_{16}$.

2.2 Représentation en machine

Dans une machine, on utilise l'écriture binaire pour représenter un entier naturel.

Avec des octets, soit huit bits, on peut représenter les entiers naturels de 0 (00000000 en base deux) à 255 (1111 1111 en base deux). Donc 45 est représenté par 00101101.

Si on utilise seize bits, soit deux octets, on peut représenter les entiers naturels jusqu'à 65535 (1111 1111 1111 1111 en base deux) et dans ce cas 45 est représenté par 0000000000101101.

Avec n bits, on peut représenter les nombres entre 0 et $2^n - 1$, c'est-à-dire tout nombre k qui s'écrit :

$$k = \sum_{i=0}^{n-1} b_i 2^i \quad \text{avec } b_i \in \{0, 1\}$$

Addition de nombres binaires

Pour additionner deux nombres en binaire on procède comme en base dix.

À partir de $0 + 0 = 0$, $1 + 0 = 0 + 1 = 1$ et $1 + 1 = 10$, on pose l'addition comme, avec le système de retenue.

Par exemple :

	1	1	0	1	treize
+	1	0	0	1	neuf
	1		1		retenues
1	0	1	1	0	vingt-deux

Attention, si la taille des entiers est limitée, par exemple avec quatre bits, alors dans l'addition ci-dessus le bit à gauche est perdu. Donc pour la machine, la somme de 1101 et de 1001 vaut 0110. Autrement dit, si nous demandons à la machine de calculer $13 + 9$, elle nous répond 6 !

2.3 Entiers relatifs

L'idée d'utiliser un bit pour le signe et les autres bits pour la valeur absolue a de nombreux inconvénients en particulier pour effectuer des opérations.

Avec quatre bits, par exemple, le bit à gauche est perdu et seize, (1 0000), est équivalent en machine à 0 (0 0000).

Donc sur quatre bits, l'opposé r' d'un nombre r s'obtient par la différence $r' = 16 - r = 2^4 - r$ puisque $r + r' = 2^4$ ($= 0$ pour la machine).

De manière générale, avec n bits, la représentation en machine d'un entier négatif r, est l'écriture binaire de la différence $2^n - r$. Cette représentation s'appelle le complément à 2^n.

En pratique, pour trouver la représentation d'un entier négatif r, on prend l'écriture binaire de $-r$ (qui est un entier positif), on inverse les bits de cette écriture et on ajoute 1.

Dans toutes ces écritures, le nombre de bits utilisés est fixé.

Exemple avec 6 bits, pour obtenir le codage de -12 :

001100 (codage de 12 en binaire sur 6 bits)

110011 (complément à un, on inverse chaque bit)

110100 (on ajoute 1) : représentation de -12 en complément à deux sur 6 bits.

Avec n bits, on peut représenter les 2^n nombres compris entre -2^{n-1} et $2^{n-1} - 1$. Les nombres négatifs ont tous le bit de poids fort égal à 1.

Les écritures binaires des entiers naturels de 0 à $2^{n-1} - 1$ représentent les entiers relatifs positifs ou nul correspondants. Les écritures binaires des entiers naturels de 2^{n-1} à $2^n - 1$ représentent les entiers relatifs négatifs de -2^{n-1} à -1.

Ainsi, si la machine utilise un octet, soit huit bits, on peut écrire les entiers naturels n de 0 à 255 et donc représenter les entiers relatifs de $-2^7 = -128$ à $2^7 - 1 = 127$. Les nombres n de 0 à 127, (de 0000 0000 à 0111 1111 en base deux), servent à représenter les entiers relatifs positifs ou nul r avec $n = r$ et les nombres n de 128 à 255, (de 1000 0000 à 1111 1111 en base deux), représentent les entiers relatifs négatifs r avec $n = r + 256$.

Remarque 1 : quel que soit le nombre n de bits utilisés, le nombre -1 est représenté par le nombre $2^n - 1$ écrit en binaire, donc par une suite de 1.

Remarque 2 : nous avons un principe équivalent sur le cercle trigonométrique gradué de degré en degré. Si nous le parcourons dans le sens direct, chaque graduation correspond à un nombre positif, dans l'ordre 0, 1, 2, …, 178, 179, 180, 181, …, 358, 359 (ceci correspond aux entiers naturels). Chaque graduation peut aussi correspondre à un entier relatif, dans l'ordre 0, 1, 2, …, 178, 179, -180, -179, …, -3, -2, -1. Ce principe se trouve aussi dans la lecture de l'heure. Nous pouvons dire "dix heures quarante" ou "onze heures moins vingt". Et cinquante-neuf correspond à "moins une".

2.4 Programmation

Avec certains langages, le nombre d'octets avec lequel on travaille sur les entiers peut être précisé (un octets, deux octets, quatre octets, huit octets).

En langage Python, la taille gérée par le langage est à priori illimitée. Les entiers sont représentés par le type **int** (integer). Cependant, si un nombre dépasse la taille maximale utilisable par le processeur, les calculs sont ralentis puisque ce nombre est découpé en deux ou plusieurs parties par Python qui s'occupe des différentes opérations à effectuer.

Il est facile de vérifier que les calculs avec des entiers longs prennent en Python plus de temps qu'avec des nombres flottants par exemple. Cependant ces calculs sont exacts !

La plupart des processeurs calculent avec des tailles limitées à 32 ou 64 bits. Les langages de programmation en général font de même et les calculs sont gérés directement par le processeur.

Pour simplifier, supposons que les entiers sont codés sur deux octets. Le langage nous permet le choix de travailler uniquement sur des entiers naturels, appelés entiers non signés (sans signe). Nous avons alors à notre disposition les nombres de 0 à 65535. Si notre programme est amené à demander le calcul $65500+40$, le reste du programme risque d'être fort compromis puisque le résultat de l'addition sera 4. (En binaire, le résultat est 1 0000… 0100 et le 1 à gauche disparaît). De même le résultat du produit 32×2048 sera 0. Avec des entiers relatifs, appelés entiers signés, toujours sur deux octets, nous disposons des nombres de -32768 à 32767. Donc, par exemple, le calcul $32000 + 1000$ aura pour résultat -32536.

Le programme ci-dessous, écrit en C++, permet de constater ces résultats.

```
#include <iostream>
using namespace std;

int main(){
    unsigned short a;
    a = 65500 + 40;
    cout<<"Valeur de a: "<<a<<endl;
    a = 32 * 2048;
    cout<<"Valeur de a: "<<a<<endl;
    short b;
    b = 32000 + 1000;
    cout<<"Valeur de b: "<<b<<endl;
    return 0;
}
```

3 Booléens

En Python une variable de type `bool` ne peut prendre que deux valeurs `True` et `False`, vrai et faux.

Une expression booléenne est composée de booléens et d'opérateurs. Nous utilisons dans la suite les opérateurs logiques AND, OR et NOT, soit ET, OU et NON.

Pour l'opérateur ET, nous obtenons les résultats présentés dans le tableau suivant que nous pouvons résumer par : (a ET b) est vrai si et seulement si a est vrai et b est vrai.

b\a	0	1
0	0	0
1	0	1

En langage Python, l'expression a ET b s'écrit `a and b`. Remarquons que si nous remplaçons les valeurs `True` et `False` respectivement par `1` et `0`, l'expression `a and b` correspond à `a * b`, c'est-à-dire le produit.

L'opérateur `&` agit sur les bits, un par un. Par exemple `b1b2b3 & b4b5b6` a pour valeur `(b1 & b4)(b2 & b5)(b3 & b6)`.

Donc `6 & 3` a pour valeur 2, car en binaire `110 & 011` a pour valeur `010`.

Pour l'opérateur OU, nous obtenons les résultats présentés dans le tableau suivant et résumés par : (a OU b) est faux si et seulement si a est faux et b est faux.

b\a	0	1
0	0	1
1	1	1

En langage Python, l'expression a OU b s'écrit `a or b`. Si nous remplaçons les valeurs `True` et `False` respectivement par `1` et `0`, l'expression `a or b` correspond à `a + b - a * b`.

L'opérateur `|` agit sur les bits, un par un. Par exemple `b1b2b3 | b4b5b6` a pour valeur `(b1 | b4)(b2 | b5)(b3 | b6)`.
Donc `6 | 3` a pour valeur 7, car en binaire `110 | 011` a pour valeur `111`.

Pour l'opérateur NON, le résultat est simple : (NON a) est vrai si et seulement si a est faux. En langage Python, l'expression NON a s'écrit `not a`.
L'opération $\sim$`a` inverse les valeurs des bits du nombre `a`.
Quel que soit le nombre de bits utilisés, $\sim$`0...0` vaut `1...1`, soit en décimal : $\sim$`0` vaut -1.

Pour les tests, les opérateurs mathématiques de comparaison utilisés sont : ==, !=, <, <=, >, >=.

Un nombre n'est pas égal au caractère qui le représente. Mais Python reconnaît si un entier et un flottant ont la "même valeur".

```
>>> 5 == "5"
False
>>> 5 == 5.0
True
```

Dans une expression booléenne, les opérandes peuvent être de différents types. Pour Python, ce qui est nul ou vide peut être interprété comme `False`, (les nombres 0 et 0.0, les chaînes de caractères vides, etc), le reste comme `True`. Certains langages n'ont pas de booléens et utilisent seulement 1 pour `True` et 0 pour `False`.

```
>>> if 2 and 5:
        print("c'est vrai")

c'est vrai
```

Séquentialité des opérateurs and et or

Dans une expression `a and b`, `a` est évalué en premier. Si `a` est faux, l'expression prend la valeur de a, sinon elle prend la valeur de `b`.

Dans une expression `a or b`, `a` est évalué en premier. Si `a` est vrai, l'expression prend la valeur de a, sinon elle prend la valeur de `b`.

Donc la valeur d'une expression `a and b`, ou `a or b`, est la dernière valeur évaluée.

```
>>> 0 and 5
0
>>> 2 and 5
5
>>> 2 and 0
0
```

```
>>> 0 or 5
5
>>> 2 or 0
2
>>> 2 or 5
2
```

L'expression `not a` ne peut avoir pour valeur que `False` ou `True`.

Table de vérité

Dans cette table, F signifie `False` et T signifie `True`. Nous résumons ici les principaux résultats pour des expressions booléennes simples.

a	b	not a	not b	a and b	a or b	not a and not b	not a or not b
T	T	F	F	T	T	F	F
T	F	F	T	F	T	F	T
F	T	T	F	F	T	F	T
F	F	T	T	F	F	T	T

Nous pouvons déduire, en comparant les valeurs de `not a and not b` avec celles de `a or b`, que `not a and not b` est équivalente à `not (a or b)`, et en comparant les valeurs de `not a or not b` avec celles de `a and b`, que `not a or not b` est équivalente à `not (a and b)`.

Ces deux équivalences s'appellent les lois de Morgan.

Un autre opérateur joue un rôle majeur dans le calcul booléen et le fonctionnement d'une machine. Il s'agit de l'opérateur logique XOR, le OU exclusif :

`a XOR b` est équivalent à `(a ET (NON b)) OU ((NON a) ET b)`.

Considérons a et b, deux chiffres binaires ou deux booléens, et les deux variables `s = a XOR b` et `r = a ET b`. La valeur de `s` représente la somme sur un bit et la valeur de `r` la retenue sur 1 bit.

Nous avons vu des opérateurs qui s'utilisent sur les bits : $\sim$ (inversion des bits), `&` (et logique), `|` (ou logique). Nous disposons aussi de l'opérateur `^` (ou exclusif).

```
>>> 1 & 1
1
>>> 1 & 0
0
>>> 0 & 1
0
>>> 0 & 0
0
```

Dans chaque cas, l'expression `a & b` a pour valeur la retenue dans l'addition `a+b`.

De même, l'expression `a ^ b` a pour valeur le chiffre des unités dans l'addition `a+b`.

```
»> 1 ^ 1
0
»> 1 ^ 0
1
»> 0 ^ 1
1
»> 0 ^ 0
```

```
    0
```

L'opérateur `^` s'utilise sur les bits mais il n'y a pas d'opérateurs logiques pour XOR comme `and` pour ET. Une possibilité est d'écrire une fonction `xor` qui prend en paramètres deux variables `a` et `b` et renvoie `a XOR b`. Contrairement aux opérateurs `and` et `or`, l'évaluation de chacune des deux valeurs `a` et `b` est obligatoire dans tous les cas.

```
def xor(a, b):
    return (a and not(b)) or (not(a) and b)
```