Plan

Plan

Représenter l'information

- L'informatique est le traitement automa**tique** de l'**informa**tion.
  - automatique = réalisable par une machine
  - information = données de l'utilisateur (des nombres, du son, des images, du texte, ...)
- Pour travailler sur des données l'ordinateur a besoin de les représenter dans un langage qu'il comprend : le langage machine basé sur le système binaire.
- On appelle **codage** le procédé qui consiste à traduire les données en langage machine et **décodage** le procédé inverse.

Le bit

- Le bit est la plus petite unité de représentation de l'information.
- bit = contraction de **bi**nary digi**t** (chiffre binaire)
- Un bit a pour valeur 0 ou 1.
- Dans certains cas (voir les opérateurs logiques) on associe aussi :
  - 0 à faux
  - 1 à vrai
- Une suite de 8 bits est appelée un **octet**.
  Une suite de 4 bits est appelée un **quartet**.
- En informatique, on code les données avec des séquences de bits (ou nombres binaires).

Notation

- Pour préciser le système (binaire ou décimal) dans lequel est écrit un nombre on utilise un indice (2 ou 10).
- Exemples :
  - 1001_2 est un nombre binaire
  - 1001_{10} est un nombre décimal
- Si l'indice n'est pas précisé, le nombre est écrit dans le système décimal.

Bits de poids fort/faible

- Dans une séquence de bits $b_{n-1}b_{n-2}\ldots b_0$, on appelle :
  - bit de **poids faible**, le bit le plus à droite (b_0)
  - bit de **poids fort**, le bit le plus à gauche (b_{n-1})
  - bit de **poids** p, le bit b_p
- Par exemple, dans la séquence $\underline{1}0111\underline{0}01\underline{0}_2$:
  - Le bit de poids faible est 0.
  - Le bit de poids fort est 1.
  - Le bit de poids 3 est 0.

Unités de représentation de l'information

Basées sur les bits		
kilo-bit	Kb	10^3 bits
mega-bit	Mb	10^6 bits
giga-bit	Gb	10^9 bits
tera-bit	Tb	10^{12} bits
peta-bit	Pb	10^{15} bits
exa-bit	Eb	10^{18} bits

Basées sur les octets		
kilo-octet	Ko	10^3 octets
mega-octet	Mo	10^6 octets
giga-octet	Go	10^9 octets
tera-octet	To	10^{12} octets
peta-octet	Po	10^{15} octets
exa-octet	Eo	10^{18} octets

Quelques ordres de grandeurs :

- un document office = 10–100 Ko
- un fichier de musique = 1–10 Mo
- un DVD = 4–8 Go
- un disque dur standard = 1–10 To = 1 millier de DVDs
- quantité de données que peut stocker la NSA au data center de Bluffdale (Utah, USA) = 3–12 Eo = 3 millions de disques durs de 2 To

Capacité de codage

- Combien de séquences binaires différentes peut-on écrire
  - avec 1 bit ? 2 séquences : 0 et 1.
  - avec 2 bits ? 4 séquences : $00, 01, 10$ et 11.
  - avec 3 bits ? 8 séquences : $000, 001, 010, 011, 100, 101, 110, 111$.
  - ...
  - avec n bits ? 2^n séquences.
- Autrement dit, l'utilisation d'un bit supplémentaire permet de doubler la capacité de codage.
- Chaque séquence binaire permet ensuite de coder une valeur parmi celles que l'on souhaite représenter (caractères, nombres, couleurs, ...).
- Inversement, pour trouver le nombre de bits nécessaires au codage de v valeurs, il faut trouver la plus petite puissance de 2 supérieure à v.
  - (On obtient cette puissance, avec l'opération $\log_2(v)$.)

Exemple : Codage de l'alphabet latin

- Pour représenter les 26 caractères de l'alphabet latin, il nous faut 5 bits.
  - (Car $2^4 = 16 < 26$ et $2^5 = 32 \geq 26$.)
- On peut ensuite utiliser la représentation suivante :
  - $00000 \rightarrow \mathrm{a}, 00001 \rightarrow \mathrm{b}, 00010 \rightarrow \mathrm{c}, \ldots, 11001 \rightarrow \mathrm{z}$.

- Pour compter en binaire c'est le même principe qu'en décimal :

  $0, 1, \underbrace{10}_{2}, \underbrace{11}_{3}, \underbrace{100}_{4}, \underbrace{101}_{5}, \underbrace{110}_{6}, \underbrace{111}_{7}, \underbrace{1000}_{8}, \underbrace{1001}_{9}, \ldots$

- On a donc :
  - $1_2 = 1 = 2^0$
  - $10_2 = 2 = 2^1$
  - $100_2 = 4 = 2^2$
  - $1000_2 = 8 = 2^3$

  ...
- Plus généralement, la valeur décimale associée au bit de poids n est 2^n.
- On en déduit donc la valeur décimale d'une suite binaire $b_{n-1}b_{n-2}\ldots b_0$:

$$b_{n-1} \times 2^{n-1} + b_{n-2} \times 2^{n-2} + \ldots + b_0 \times 2^0 = \sum_{i=0}^{n-1} b_i \times 2^i$$

- Exemple :
  $1001101_2 = 1 \times 2^6 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^0 = 64 + 8 + 4 + 1 = 77$

Plan

Généralisation à une base B quelconque

- On connaît maintenant le système binaire basé sur les bits et le système décimal basé sur les chiffres arabes.
- On appelle base B un système dans lequel on dispose d'un alphabet de B chiffres (ou symboles) pour écrire des nombres.
  - système binaire = base 2
  - système décimal = base 10
- Ce que l'on a vu pour le système binaire peut être généralisé :
  - $x_{n-1}x_{n-2}\ldots x_0$ dans une base B vaut $x_{n-1} \times B^{n-1} + x_{n-2} \times B^{n-2} + \ldots + x_0$ en décimal.
  - Dans une base B, n chiffres permettent de coder B^n valeurs différentes.
- Exemples :
  - $5672_8 = 5 \times 8^3 + 6 \times 8^2 + 7 \times 8^1 + 2 \times 8^0 = 3002$
  - En base 7 on peut coder, avec 3 chiffres, $7^3 = 343$ valeurs différentes.

Les bases courantes

B	Nom	Chiffres	Exemple : écriture de 93
2	binaire	$0, 1$	1011101_2
8	octale	$0, 1, 2, 3, 4, 5, 6, 7$	135_8
10	décimale	$0, 1, 2, 3, 4, 5, 6, 7, 8, 9$	93_{10}
16	hexadécimale	$0, 1, 2, 3, 4, 5, 6, 7,$ $8, 9, A, B, C, D, E, F$	$5D_{16}$

Exemples d'utilisations :

- binaire : par l'ordinateur pour faire des calculs
- octale : pour représenter les droits sur un fichier (lecture, écriture, exécution)
- hexadécimale : pour représenter des séquences de bits de manière compacte

Base 10 $\rightarrow$ Base B

Problème : convertir un nombre décimal N en un nombre d'une base B

- On cherche à décomposer N en puissances de B :

$$N = r_{n-1} \times B^{n-1} + \ldots + r_0 \times B^0$$

- Méthode générale :
  - On divise N par B.
  - Le reste forme le chiffre de poids faible.
    (Le reste est forcément dans l'intervalle $[0..B-1]$.)
  - On recommence l'opération avec le quotient.
  - On s'arrête lorsque le quotient vaut 0.

conversion de 157 en binaire

division de	**157**	par 2 :	157	$=$	2	$\times$	**78**	$+$	**1**	r_0
division de	**78**	par 2 :	78	$=$	2	$\times$	**39**	$+$	**0**	r_1
division de	**39**	par 2 :	39	$=$	2	$\times$	**19**	$+$	**1**	r_2
division de	**19**	par 2 :	19	$=$	2	$\times$	**9**	$+$	**1**	r_3
division de	**9**	par 2 :	9	$=$	2	$\times$	**4**	$+$	**1**	r_4
division de	**4**	par 2 :	4	$=$	2	$\times$	**2**	$+$	**0**	r_5
division de	**2**	par 2 :	2	$=$	2	$\times$	**1**	$+$	**0**	r_6
division de	**1**	par 2 :	1	$=$	2	$\times$	**0**	$+$	**1**	r_7

Le quotient vaut 0 $\Rightarrow$ on s'arrête

Conclusion : $157_{10} =$

1	**0**	**0**	**1**	**1**	**1**	**0**	**1**
r_7	r_6	r_5	r_4	r_3	r_2	r_1	r_0

conversion de 157 en hexadécimal

division de **157** par 16 : $157 = 16 \times \mathbf{9} + \boxed{\mathbf{13}}$ r_0

division de **9** par 16 : $9 = 16 \times \mathbf{0} + \boxed{\mathbf{9}}$ r_1

Le quotient vaut 0 $\Rightarrow$ on s'arrête

Conclusion : $157_{10} =$ **9** **D**

r_1 r_0

(13 est représenté par le caractère D en hexadécimal.)

Problème : convertir un nombre d'une base B en un nombre décimal

- Il suffit de décomposer le nombre en fonction des puissances de la base et d'additionner.
- Exemples :
  - $1011101_2 = 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 93_{10}$
  - $E07_{16} = 14 \times 16^2 + 0 \times 16^1 + 7 \times 16^0 = 3\ 591_{10}$

Problème : convertir un nombre binaire en un nombre hexadécimal

- Avec 4 bits on peut coder $2^4 = 16$ valeurs = 1 chiffre hexadécimal.
- Méthode générale :
  - On groupe les bits par 4 en commençant par les bits de poids faible.
  - On rajoute si nécessaire des bits de poids fort à 0 pour obtenir un nombre de bits multiple de 4.
  - On convertit chaque suite de 4 bits en hexadécimal comme suit :

0000	⇒	0	0100	⇒	4	1000	⇒	8	1100	⇒	*C*
0001	⇒	1	0101	⇒	5	1001	⇒	9	1101	⇒	*D*
0010	⇒	2	0110	⇒	6	1010	⇒	*A*	1110	⇒	*E*
0011	⇒	3	0111	⇒	7	1011	⇒	*B*	1111	⇒	*F*

- Exemples :
  - $10110011_2 = 1011\ 0011_2 = B3_{16}$
  - $11100_2 = 0001\ 1100_2 = 1C_{16}$

Problème : convertir un nombre hexadécimal en un nombre binaire

- Il suffit de coder chaque chiffre hexadécimal en une suite de 4 bits.
- Exemples :
  - $A7_{16} = 1010\ 0111_2$
  - $7EF_2 = 0111\ 1110\ 1111_{16}$

Plan

L'addition

- L'addition se fait (quelle que soit la base) comme en base 10.
- Exemple : addition de $149 = 10010101_2$ et $305 = 100110001_2$ en binaire

			1	1				1		retenues
	0	1	0	0	1	0	1	0	1	
+	1	0	0	1	1	0	0	0	1	
	1	1	1	0	0	0	1	1	0	

On trouve bien $111000110_2 = 454 = 149 + 305$

Le dépassement de capacité

- Avant de réaliser une opération, l'ordinateur stocke les opérandes dans des **registres** (zones mémoire très rapides d'accès).
- Ces registres ont des tailles limitées (8, 32 ou 64 bits par exemple).

⇒ L'intervalle des valeurs que l'on peut y stocker est limité.

- Si le résultat d'une opération ne tient pas dans le registre, certains bits seront « oubliés » et le résultat sera incorrect.
- On appelle ce phénomène **dépassement de capacité** ou **overflow**.
- Exemple : addition sur 8 bits de $77 = 01001101_2$ et $201 = 11001001_2$.

	1	1			1			1	retenues
		0	1	0	0	1	1	0	1
+		1	1	0	0	1	0	0	1
	~~1~~	0	0	0	1	0	1	1	0

⇒ Lorsque l'ordinateur effectue $77_{10} + 201_{10}$ avec des registres de 8 bits, il trouve $00010110_2 = 22_{10}$.

- Le dépassement de capacité est la source de nombreux bugs.
  Exemple : crash d'ariane 5 en 1996 dû à l'utilisation de registres de 16 bits pour des opérations nécessitant 64 bits.

Opérateurs logiques

- En plus des opérateurs arithmétiques $(+, -, \times, /)$ les programmes ont parfois besoin d'effectuer des opérations logiques.
- On ne voit plus alors la séquence binaire comme un nombre mais comme une suite de valeurs **vrai** (1) ou **faux** (0).
- Les opérateurs logiques sont des opérateurs bit à bit : dans le résultat, le bit de poids p dépend uniquement du (ou des) bit(s) de poids p dans le(s) opérandes.
- On utilise pour cela une **table de vérité** qui donne, en fonction des bits des opérandes, le bit résultant de l'opération.
- Nous allons voir 4 opérateurs logiques : ET, OU, OU exclusif et NON.
- Exemples d'utilisation de ces opérateurs :
  - ET, OU et NON utilisés pour certains calculs sur les adresses IP (voir cours et TD 2)
  - OU exclusif utilisé par des algorithmes de cryptage (voir TD 1)

Le ET logique

- Les deux bits doivent être à vrai.
- Table de vérité du ET :

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

- Exemple :

	1	1	0	1	1	0	0	1
ET	0	1	1	1	0	0	1	1
	0	1	0	1	0	0	0	1

Le OU logique

- Au moins une des opérandes doit être à vrai.
- Table de vérité du OU :

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

- Exemple :

	1	1	0	1	1	0	0	1
OU	0	1	1	1	0	0	1	1
	1	1	1	1	1	0	1	1

Le OU excusif (XOR)

- Une des deux opérandes doit être à vrai mais pas les deux.
- Table de vérité du XOR :

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

- Exemple :

	1	1	0	1	1	0	0	1
XOR	0	1	1	1	0	0	1	1
	1	0	1	0	1	0	1	0

Le NON logique

- Inversion du bit de l'opérande.
- Table de vérité du NON :

a	**NON** a
0	1
1	0

- Exemple :

NON	0	1	1	1	0	0	1	1
	1	0	0	0	1	1	0	0

Complétude Fonctionelle

- Certains connecteurs logiques sont en réalité redondant.
- Un ensemble de connecteurs logiques est fonctionnellement complet si ces connecteurs sont suffisant pour écrire toute fonction binaire.
- Exemples :
  - $\{\text{ET}, \text{NON}\}$
  - $\{\text{OU}, \text{NON}\}$

Exercice

Écrire en fonction de ET et NON tous les autres opérateurs.

Comment représenter du texte ?

- Une **table de codage** associe une séquence binaire à chaque caractère de l'alphabet que l'on veut coder.
- L'ordinateur traduit ensuite chaque caractère du texte en la séquence de bits correspondante en utilisant cette table.
- Il y a différentes tables de codage.
- Elles se différencient par :
  - l'ensemble des caractères qu'elles permettent de coder
  - et le nombre de bits qu'elles utilisent pour coder les caractères.
- Nous allons en voir 2 : l'ASCII et l'UTF-8.
- Pour voir le codage utilisé pour un fichier : `file <nom-du-fichier>`.
- Exemple :

```
$ file message.txt
message.txt: UTF-8 Unicode text
```

Le codage ASCII

- ASCII = American Standard Code for Information Interchange
- caractères ASCII codés sur 7 bits
- Permet de coder : *les lettres de l'alphabet latin* (minuscules et majuscules), *les chiffres arabes*, *les caractères de ponctuation* et d'autres *caractères spéciaux* (ex : +, -, ESC (la touche Echap du clavier)).
- La table des caractères ASCII :

	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	SPC	!	¨	#	$	%	&	`	(	)	*	+	,	-	.	/
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

- En ligne : les 3 bits de poids fort. En colonne : les 4 bits de poids faible.
- Exemple : le caractère E est codé $\underbrace{100}_{4}\ \underbrace{0101}_{5}$

Le codage UTF-8

- UTF-8 = Universal character set Transformation Format - 8 bits
- codage le plus utilisé dans les systèmes Linux
- caractères UTF-8 codés sur 1, 2, 3 ou 4 octets
- permet de coder des textes d'à peu près n'importe quel alphabet (arabe, chinois, grec, indien, latin, ...)
- compatible avec l'ASCII : Un texte codé en ASCII est codé de la même manière en UTF-8.
  - (Les caractères ASCII étant codés sur 7 bits on rajoute un bit de poids fort à 0 pour obtenir un caractère UTF-8 sur 1 octet.)
- Exemples de codage :

$$
\begin{array}{lcl}
\text{E} & \rightarrow & 01000101 \\
\text{é} & \rightarrow & 11000011\ 10101001 \\
\text{€} & \rightarrow & 11100010\ 10000010\ 10101100 \\
\alpha & \rightarrow & 11001110\ 10110001
\end{array}
$$