

# L1S2 Théorie de l'Information

Université de Toulouse

Départements de Mathématique et d'Informatique

Année 2016/2017

# Plan

- 1 Codage
- 2 La notion d'information selon Shannon et codage optimal
- 3 Complexité de Kolmogorov et compression de données
- 4 Codes correcteurs et détecteurs d'erreurs

# Plan

- 1 Codage
  - Motivation
  - Codage de caractères
  - Codes

# Modèle de communication (1)

Modèle naïf :



**Question :** Comment transmettre l'information ?

# Modèle de communication (2)

Modèle avec canal :

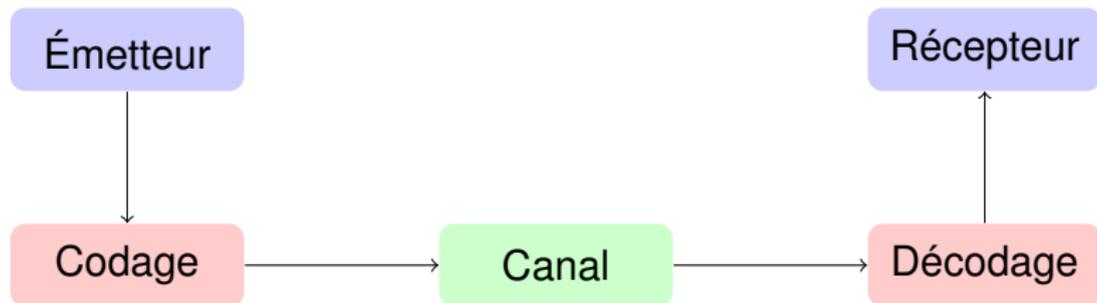


Questions :

- Quels types de données sont transmissibles ?  
*Ex. : comment transmettre une image ?*
- Capacité du canal ?
- Canal bruité ?
- Confidentialité du canal ?

# Modèle de communication (3)

## Modèle avec codage :

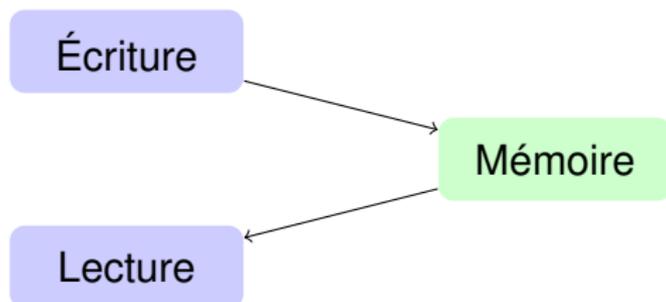


## Solutions ... et nouvelles questions :

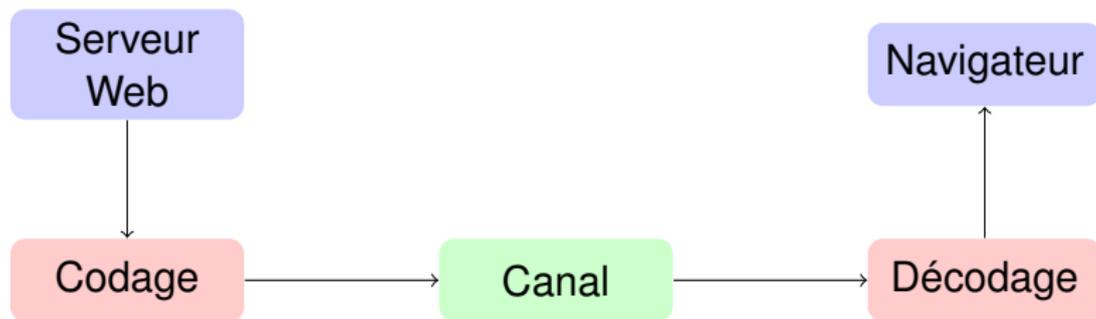
- Codage de données (*Ex* : nombres ; caractères ; images ; son)  
*Réversibilité* : peut-on reconstruire le message original ?
- Compression de données. *Optimalité* ?
- Détection et correction d'erreurs. *Pour quelle perturbation* ?
- Cryptage. *Quelle sécurité pour quel type d'adversaire* ?

# Stockage de données

comme cas particulier de communication :



# Modèle client-serveur



page HTML :

```
<ul>
<li> Une <b>liste</b> </li>
<li> et une <i>balise</i> </li>
</ul>
```

affichage navigateur :

- Une **liste**
- et une *balise*

# Plan

- 1 Codage
  - Motivation
  - Codage de caractères
  - Codes

# Codage de caractères : Précurseurs

## Code Morse

- Développé par Samuel Morse et collaborateurs (1837)
- Transmission de signaux par télégraphe
- Codage de caractères par séquences signal court / long



## Code morse international

1. Un trait est égal à trois points.
2. L'espacement entre deux éléments d'une même lettre est égal à un pt
3. L'espacement entre deux lettres est égal à trois points.
4. L'espacement entre deux mots est égal à sept points.

A	• —	U	• • —
B	— • • •	V	• • — —
C	— • — • •	W	— • • —
D	— • • •	X	— • — — —
E	•	Y	— • • — — —
F	• • — • •	Z	— — • • •
G	— • — • •		
H	• • • •		
I	• •		
J	• — — — —	1	• — — — — —
K	— • — — —	2	• • — — — —
L	— • • •	3	• • • — — —
M	— — —	4	• • • • — —
N	— • •	5	• • • • •
O	— — — —	6	— • • • •
P	— • — • •	7	— — • • • •
Q	— — — — —	8	— • — • • •
R	• — • •	9	— — — • • •
S	• • •	0	— — — — — •
T	—		

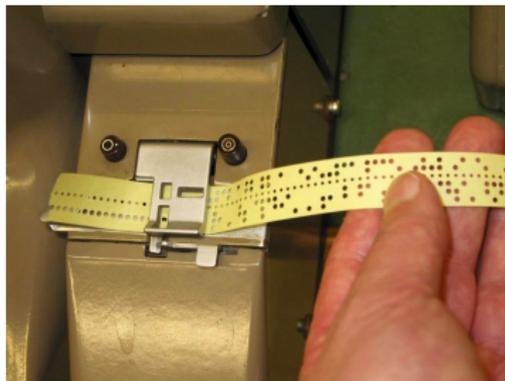
# Codage de caractères : Précurseurs



## Téléscripteurs

- Machines à écrire à distance
- Premiers développements  $\approx$  1930
- Caractéristiques :
  - Alphabet restreint (caractères non accentués, majuscules)
  - Affichage sans écran !
  - Codage du mouvement du chariot de la machine (saut de ligne, retour chariot)
  - Stockage de données sur ruban perforé

# Codage de caractères : Précurseurs



## Code Baudot

- Inventé en 1870 par Émile Baudot
- Système à 5 bits transmis simultanément
- Deux modes :
  - 1 26 lettres  
ex. : 01010 lettre R
  - 2 10 chiffres, quelques symboles  
ex. : 01010 chiffre 4
- Séquences de contrôle :
  - 01000 retour chariot
  - 11111 changement mode “lettres”
  - 11011 changement mode “chiffres”

# Codage de caractères : ASCII (1)

## American Standard Code for Information Interchange

- Standardisé en 1963 : norme ASA des États-Unis
- Depuis 1972 : norme ISO 646

## Codage binaire de 128 caractères en 7 bits :

- caractères de l'alphabet anglais (minuscules et majuscules)
- chiffres
- quelques symboles
- des caractères de contrôle (en partie pour téléscripteurs)

## Encore perceptible dans certaines limitations :

- adresses email
- langages de programmation

# Codage de caractères : ASCII (2)

## Caractères de contrôle

<i>Bin</i>	<i>Déc</i>	<i>Car</i>	<i>Signif.</i>
000 0000	0	NUL	End of string
000 0010	2	STX	Start of text
000 0011	3	ETX	End of text
000 1000	8	BS	Backspace
000 1001	9	HT	Horizontal Tab
000 1010	10	LF	Line feed
000 1101	13	CR	Carriage return
001 1011	27	ESC	Escape
111 1111	127	DEL	Delete

## Caractères affichables

<i>Bin</i>	<i>Déc</i>	<i>Car</i>
010 0000	32	␣
010 0001	33	!
010 1111	47	/
011 0000	48	0
011 1001	57	9
100 0001	65	A
101 1010	90	Z
110 0000	96	'
110 0001	97	a
111 1010	122	z
111 1110	126	~

Table complète : [http://fr.wikipedia.org/wiki/American\\_Standard\\_Code\\_for\\_Information\\_Interchange](http://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange)

# Transmission de données

## Anciens systèmes :

- Transmission de données “directe” :
  - terminal → imprimante
  - teletype → teletype via connexion téléphonique
- ...encadré par des caractères de contrôle :

This is a text ~↔

STX T h ... t ETX ~↔

0000010 1010100 1101000 ... 1110100 0000011

## Systèmes modernes : Transmission en paquets contenant des méta-données :

- Longueur du texte
- Destinataire, routage
- Qualité de service (priorité, prix) ; redondance (~↔ correction d'erreurs)

~↔ perte d'importance des caractères de contrôle

# Stockage de données dans des fichiers

## Fin de fichier (end of file, EOF)

- *(Quelques) anciens systèmes :*
  - *Exemple : Système CP/M (par Intergalactic Digital Research) :*  
Fichiers sont des multiples de blocs de 128 octets  
Fin de fichier indiqué par `Ctrl-Z`
- *Systèmes modernes :*
  - Taille d'un fichier stocké par le système d'exploitation (OS)
  - Il n'existe pas de caractère EOF

## Retour à la ligne Différents codages, selon le OS :

- `CR + LF` dans d'anciens téléscripateurs  
hérité par des successeurs : DEC ; CP/M, MS-DOS, Windows
- `LF` dans la famille Multics, Unix, Linux
- `CR` dans l'ancien MacOS

Voir programmes `dos2unix` et `unix2dos`

# Codage de caractères : famille ISO/IEC 8859 (1)

## Motivation :

- Codage de caractères d'autres alphabets
- Utilisation du 8ième bit d'un octet

## Représentants :

- ISO 8859-1 : Codage de la plupart des alphabets occidentaux ("Latin-1")  
*imparfaitement* : il manque œ, Œ, €, inclus dans ISO 8859-15
- ISO 8859-5 : Cyrillique
- ISO 8859-7 : Grecque
- ...



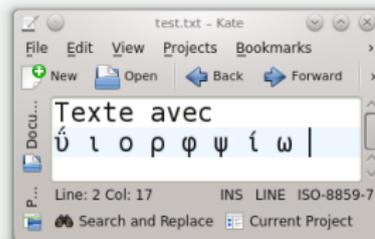
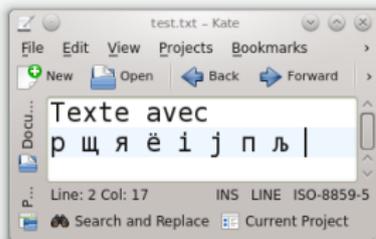
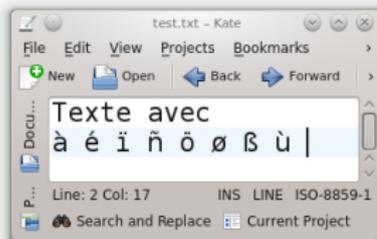
# Codage de caractères : famille ISO/IEC 8859 (3)

## Interprétation d'un fichier

- Un éditeur ne voit que le code binaire
- ...et l'interprète selon le codage choisi

## Un fichier sous trois vues

- Ici : éditeur Kate
- Changement de codage : `Tools / Encoding / ...`



# Codage de caractères : Unicode (1)

**Intention** : Pouvoir coder tous les alphabets actuels et passés

- Europe : Latin, grecque, cyrillique, arménien, géorgien, ...
- Moyen orient : hébreu, arabe, syriaque, ...
- Inde : Devanagari, bengali, gujarati, ...
- Asie : Han (Chine), Hiragana et Katakana, ...
- Afrique : Éthiopien, N'Ko (Afrique orientale), Bamum (Cameroun)
- Amérique : Cherokee (indien)
- Ancien : runique, gothique, Linear B (ancien grecque), phoenicien, hiéroglyphes égyptiens, ...
- Symboles : mathématiques ; monnaies ;

Pour des détails, voir <http://www.unicode.org/>

# Codage de caractères : Unicode (2)

## Distinction entre :

- Caractère abstrait (“code point”)
- son codage, en trois variantes : UTF-8, UTF-16, UTF-32

## Quelques chiffres :

- Unicode définit plus de 110.000 caractères
- en principe : 1.114.112 “code points” (nombres  $0 \dots 10FFFF_{16}$ )

## Développement historique :

- 1987 : Première tentative de remplacer ASCII par un codage “universel”
- 1991 : Création du Unicode Consortium (Xerox, Apple, Sun, Next, Microsoft)
- 1993 : norme ISO/IEC 10646
- sept. 2013 : version 6.3

# Codage de caractères : Unicode (3)

## Caractères abstraits, décrits par

- un *code point* (numéro d'identification)
- un glyphe (représentation graphique)
- un nom textuel

### Exemple : Latin-1 Supplement

	008	009	00A	00B	00C	00D	00E	00F
0	XXX 0080	DCS 0090	NB SP 00A0	◊ 00B0	À 00C0	Ð 00D0	à 00E0	ǵ 00F0
1	XXX 0081	PU1 0091	ı 00A1	± 00B1	Á 00C1	Ñ 00D1	á 00E1	ñ 00F1
2	BPH 0082	PU2 0092	ç 00A2	2 00B2	Â 00C2	Ò 00D2	â 00E2	ò 00F2
3	NBH 0083	STS 0093	£ 00A3	3 00B3	Ã 00C3	Ó 00D3	ã 00E3	ó 00F3

- 00E0 à LATIN SMALL LETTER A WITH GRAVE  
≡ 0061 a 0300 ð
- 00E1 á LATIN SMALL LETTER A WITH ACUTE  
≡ 0061 a 0301 ó
- 00E2 â LATIN SMALL LETTER A WITH CIRCUMFLEX  
≡ 0061 a 0302 ô
- 00E3 ã LATIN SMALL LETTER A WITH TILDE  
• Portuguese  
≡ 0061 a 0303 õ

# Codage de caractères : Unicode (4)

## Autres alphabets :

### Devanagari

	090	091	092	093
0	ॐ 0900	ऐ 0910	ठ 0920	र 0930
1	ँ 0901	ऑ 0911	ड 0921	ॠ 0931

090F	ए	DEVANAGARI LETTER E
0910	ऐ	DEVANAGARI LETTER AI
0911	ऑ	DEVANAGARI LETTER CANDRA O
0912	ओ	DEVANAGARI LETTER SHORT O
		• for transcribing Dravidian short o
0913	ओ	DEVANAGARI LETTER O
0914	औ	DEVANAGARI LETTER AU

### Egyptian Hieroglyphs

	130E	130F	1310	1311
0	 130E0	 130F0	 13100	 13110
1	 130E1	 130F1	 13101	 13111

130E0		EGYPTIAN HIEROGLYPH E013
130E1		EGYPTIAN HIEROGLYPH E014
130E2		EGYPTIAN HIEROGLYPH E015
130E3		EGYPTIAN HIEROGLYPH E016
130E4		EGYPTIAN HIEROGLYPH E016A

## Entrer un caractère dans Kate :

*Exemple* : F7, ensuite char 0x90f

# Codage de caractères : Unicode (5)

Trois codages pour chaque caractère Unicode :

A	Ω	語	𐄎	UTF-32
00000041	000003A9	00008A9E	00010384	

A	Ω	語	𐄎	UTF-16
0041	03A9	8A9E	D800   DF84	

A	Ω	語	𐄎	UTF-8
41	CE   A9	E8   AA   9E	F0   90   8E   84	

# Codage de caractères : Unicode (6)

**UTF-32** Chaque caractère représenté par un mot de 32 bits

Code point	Code (hex)	Caractère	Nom
U+41	00000041	A	Latin capital letter a
U+E1	000000E1	á	Latin small letter a with acute
U+910	00000910		Devanagari letter ai
U+130E0	000130E0		Egyptian hieroglyph E013

## Avantages :

- Codage uniforme et simple

## Désavantages :

- Utilisation d'une petite fraction de l'espace des codes

# Codage de caractères : Unicode (7)

**UTF-16** Un ou deux mots de 16 bits, construits comme suit :

- ① U+0000 ... U+D7FF et U+E000 ... U+FFFF :  
représentés par *un* mot de 16 bits avec la même valeur

Code point	Code (hex)	Caractère	Nom
U+41	0041	A	Latin capital letter a
U+910	0910		Devanagari letter ai

- ② U+D800 ... U+DFFF : ne sont pas des code points valides
- ③ U+10000 ... U+10FFFF : deux mots.

Algorithme pour conversion de  $U+x_{16}$  :

- ① Calculer  $(x')_{16} = (x)_{16} - (10000)_{16}$
- ② Représenter  $(x')_{16}$  en binaire  $(b')_2$  avec 20 chiffres :  $(x')_{16} = (b')_2$
- ③ Scinder  $(b')_2$  en deux mots  $v$  et  $w$  de 10 bits
- ④ Résultat du codage :
  - Premier mot :  $(110110v)_2$
  - Deuxième mot :  $(110111w)_2$

# Codage de caractères : Unicode (8)

*Exemple* : Codage de U+10384 en UTF-16

Ici :  $(x)_{16} = (10384)_{16}$

- 1 Calculer  $(x')_{16} = (10384)_{16} - (10000)_{16} = (384)_{16}$
- 2 Représenter en binaire avec 20 chiffres :  
 $(b')_2 = 0000.0000.0011.1000.0100$
- 3 Scinder en  $v = 0000.0000.00$  et  $w = 11.1000.0100$
- 4 Résultat :
  - Premier mot :  $(1101.1000.0000.0000)_2 = (D800)_{16}$
  - Deuxième mot :  $(1101.1111.1000.0100)_2 = (DF84)_{16}$

Avantages / désavantages de UTF-16 :

- 1 Codage assez complexe
- 2 Longueur du code : compromis entre UTF-32 et UTF-8

# Codage de caractères : Unicode (9)

**UTF-8** Codage entre 1 et 4 octets, selon la table :

<i>Intervalle</i>	<i>Octet 1</i>	<i>Octet 2</i>	<i>Octet 3</i>	<i>Octet 4</i>
U+0 ... U+7F	0xxxxxxx			
U+80 ... U+7FF	110xxxxx	10xxxxxx		
U+800 ... U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+10000 ... U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Les xxx sont les chiffres d'une écriture (souvent impropre) du code point en base 2 avec 7, 11, 16 et 21 chiffres, selon le cas.

*Notes* : U+0 ... U+7F et ASCII coïncident

# Codage de caractères : Unicode (10)

## Exemples de codage en UTF-8

- $U+41 \in U+0 \dots U+7F$ , donc codage avec 1 octet  
 $(41)_{16} = (100.0001)_2$   
 $\rightsquigarrow (0100.0001)_2 = (41)_{16}$
- $U+3A9 \in U+80 \dots U+7FF$ , donc codage avec 2 octets  
 $(3A9)_{16} = (011.1010.1001)_2$   
 $\rightsquigarrow (1100.1110.1010.1001)_2 = (CEA9)_{16}$

# Codage de caractères : Unicode (11)

## Comment reconnaître le codage d'un fichier stocké sur disque ?

- Impossible : deux sources différentes peuvent avoir le même codage
- Il existe des heuristiques ...

## ...transmis par internet ?

- Pareil ...
- Mais : voir les méta-données de certains protocoles.  
*Exemple* : HTML (visualiser source avec `Ctrl-u`)

```
<!DOCTYPE html>
<html dir="ltr" lang="fr" xml:lang="fr">
<head>
  <title>Cours: L1S2 Math-Info 2</title>
  <meta http-equiv="Content-Type"
    content="text/html; charset=utf-8" />
```

# Résumé : Codage de caractères

## Histoire du codage de caractères :

- Développement successif au lieu de rupture technologique
- Intégration de caractères de plus en plus complexes : maj./minuscules ; accents ; internationalisation

## Unicode

- Distinction entre “caractère” et “codage”
- Principe de UTF-16 et UTF-8 : code “plus court” pour caractères “plus utilisés”

## Question :

Existe-t-il un codage optimal pour une quantité d'information ?

# Annexe : Codage de nombres (1)

**Base 2** : Chiffres 0, 1

**Base 10** : Chiffres 0 .. 9

**Base 16** : Chiffres 0 .. 9, A .. F

**Codage** en base  $b$  d'un nombre  $n$  :

Séquence de chiffres  $c_k \dots c_0$  tels que  $n = c_k * b^k + \dots + c_0 * b^0$

*Exemple* :  $(13)_{10} = 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = (1101)_2$

*Exemple* :  $(42)_{10} = 2 * 16^1 + 10 * 16^0 = (2A)_{16}$

**Conversion rapide** base 16  $\leftrightarrow$  base 2 :

Regroupement en paquets de 4 chiffres :

*Exemple* :  $(F2A)_{16}$

$$= 15 * 16^2 + 2 * 16^1 + 10 * 16^0$$

$$= 15 * (2^4)^2 + 2 * (2^4)^1 + 10 * (2^4)^0$$

$$= ((1111)(0010)(1010))_2$$

## Annexe : Codage de nombres (2)

Pour  $b > 0$ , tout nombre  $n$  peut s'écrire  $n = (n/b) * b + (n \bmod b)$ , où  $/$  est la division entière et  $\bmod$  le reste de la division.

*Exemple* :  $7/3 = 2$  et  $7 \bmod 3 = 1$ .

Ceci motive la **fonction de codage**  $\text{coder}(\text{nombre}, \text{base})$  définie par :

- si  $n < b$  :  $\text{coder}(n, b) = n$
- si  $n \geq b$  :  $\text{coder}(n, b) = \text{coder}(n/b, b) * b + (n \bmod b)$

*Exemple* :

$$\text{coder}(380, 16) =$$

$$\text{coder}(380/16, 16) * 16 + (380 \bmod 16) =$$

$$\text{coder}(23, 16) * 16 + 12 =$$

$$(\text{coder}(23/16, 16) * 16 + (23 \bmod 16)) * 16 + 12 =$$

$$(\text{coder}(1, 16) * 16 + 7) * 16 + 12 =$$

$$(1 * 16 + 7) * 16 + 12 = 1 * 16^2 + 7 * 16^1 + 12 * 16^0$$

$$\text{Donc, } (380)_{10} = (17C)_{16}$$

# Annexe : Commandes Linux

## ll list directory contents

```
> ll test.txt
```

```
-rw-r--r-- 1 strecker users 29 Jan 23 01:54 test.txt
```

## wc print newline, word, and byte counts for each file

```
> wc main.tex
```

```
119 252 3251 main.tex
```

## xxd make a hexdump

```
> xxd -c 10 test.txt
```

```
00000000: 5465 7874 6520 6176 6563  Texte avec
0000000a: 0ac3 a020 c3a9 20c3 af20  ... ..
00000014: c3b1 20c3 b620 c3b8 20c3  .. .. .
0000001e: 9f20 c3b9 0a                . ...
```

# Plan

- 1 Codage
  - Motivation
  - Codage de caractères
  - Codes

# Alphabets, séquences et messages

Un **alphabet**  $A$  est un ensemble de caractères.

*Exemples* : Alphabet

- des caractères ASCII
- des caractères de Unicode
- des chiffres 0 et 1 (alphabet binaire)
- des chiffres 0 ... 9, A ... F (hexadécimal)

Une **séquence** est inductivement construite comme suit :

- $[]$  est une séquence (vide)
- si  $a \in A$  est un caractère et  $s$  est une séquence, alors  $a \cdot s$  est une séquence. (On omet souvent le constructeur  $\cdot$ )

*Notation* :  $A^*$  est l'ensemble de séquences sur  $A$

Un **message** (en  $A$ ) est une séquence de caractères de  $A$ .

# Codage et décodage

**Codage** : fonction  $c^* : A_1^* \rightarrow A_2^*$ , pour deux alphabets  $A_1$  et  $A_2$ .  
 Le *décodage* est un codage ... en sense inverse :  $d^* : A_2^* \rightarrow A_1^*$   
 Un codage  $c^*$  est

- **unique / sans perte** s'il existe un décodage  $d^*$  tq. pour tout message  $m$ ,  $d^*(c^*(m)) = m$   
*Fortement souhaitable* pour tout codage de textes, nombres, ...
- *avec perte* sinon  
*Acceptable* pour du son, des images ...

*Exemple :*

- codage  $c^*(xyz) = 78797A$ , pour  $c^* : UTF8^* \rightarrow HEX^*$
- décodage  $d^*(78797A) = xyz$ , pour  $d^* : HEX^* \rightarrow UTF8^*$

# Codes et codages

Un **code**  $c : A_1 \rightarrow A_2^+$  traduit un caractère de  $A_1$  vers une séquence de caractères  $A_2$  non-vide.

*Exemple* :  $c(x) = 78$ ,  $c(y) = 79$ ,  $c(z) = 7A$

Un codage  $c^* : A_1^* \rightarrow A_2^*$  est l'**extension homomorphe** du code  $c$  s'il est défini par

$$c^*(x_1 x_2 \dots x_n) = c(x_1)c(x_2) \dots c(x_n)$$

*Exemple* :  $c^*(xyz) = c(x)c(y)c(z) = 78797A$

*Contre-exemple 1* :

- $d^*(78797A) \neq d(7)d(8) \dots$
- *Pourtant* :  $d^*(78797A) = d^*(78)d^*(79)d^*(7A)$

# Codes injectifs et préfixes

*Définition* : Une fonction  $f$  est **injective** si pour tout  $m_1, m_2$ , si  $m_1 \neq m_2$ , alors  $f(m_1) \neq f(m_2)$

Ainsi, on parle d'un **code injectif**  $c$  et d'un **codage injectif**  $c^*$ .

*Définition* :  $m$  est un **préfixe** de  $m'$  s'il existe  $r$  avec  $m' = mr$

*Notation* :  $m \preceq m'$

*Exemples* :  $01 \preceq 011$ ,  $011 \preceq 011$ ,  $010 \not\preceq 011$

$c$  est un **code préfixe** si pour tout  $a, a' \in A$  avec  $a \neq a'$  :  $c(a) \not\preceq c(a')$

# Codages préfixes

Désormais : codages *homomorphes* et *préfixes*

Comment **coder** un message ?

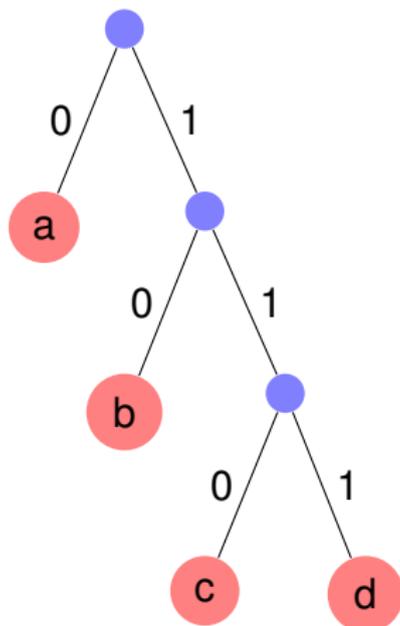
Facile (comme pour tout codage homomorphe)

Ex. : Codez *acdc* selon la table :

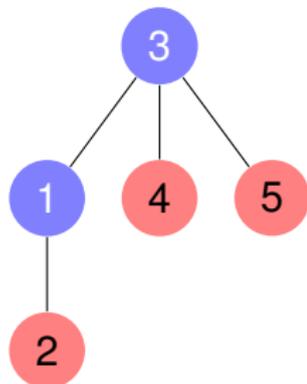
$x$	$c(x)$
a	0
b	10
c	110
d	111

Comment **décoder** un message ?

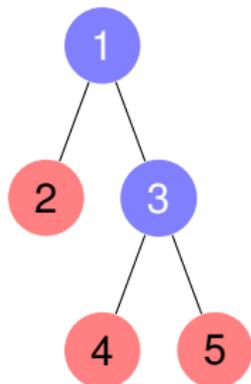
Ex. : Décodez 100110



# Interlude : Définitions d'arbres



?



Plusieurs définitions possibles :

- Graphe avec un chemin unique entre deux noeuds
- Graphe connecté qui devient déconnecté si on supprime un arc
- ...
- **ici** : Définition *inductive*. Avantages :
  - Traitement facile par *réursion*
  - Prototype de beaucoup de structures informatiques : expressions, programmes, ...

# Interlude : Construction inductive d'un arbre

## Conventions :

- Arbres binaires : chaque noeud intérieur a exactement 2 successeurs
- On peut associer une information au noeuds ou feuilles

## Construction inductive :

- 1 Une feuille est un arbre  
*Écriture :  $L(\text{info}) : \text{arbre}$*
- 2 Un noeud intérieur avec deux sous-arbres est un arbre  
*Écriture :  $N(\text{info}, \text{arbre}, \text{arbre}) : \text{arbre}$*

...et c'est la seule manière de construire un arbre.

## Exemple :

$$N(1, L(2), \\ N(3, L(4), \\ L(5)))$$

# Interlude : Fonctions récursives sur les arbres

**Principe** de la définition d'une fonction  $f$  par récursion :

- 1 Définir le résultat de  $f$  sur une feuille :  

$$f(L(i)) = lres(i)$$
- 2 Définir le résultat de  $f$  sur un noeud, si on connaît les résultats sur les sous-arbres :  

$$f(N(i, a_1, a_2)) = nres(i, f(a_1), f(a_2))$$

**Exemple** : Somme des valeurs :

$$\text{somme}(L(i)) = i$$

$$\text{somme}(N(i, a_1, a_2)) = i + \text{somme}(a_1) + \text{somme}(a_2)$$

**Exercice** : Calculez  $\text{somme}(N(1, L(2), N(3, L(4), L(5))))$

# Construction d'un arbre de décodage

...à partir d'une table de codage, *pour un code préfixe*.

**Représentation** de la table de codage comme ensemble de couples (caractère, code associé)

*Exemple* :  $\{(a, 0); (b, 10); (c, 110); (d, 111)\}$

`arbre_dec` prend : table non vide ; construit : arbre de décodage.

`arbre_dec` ({} ) = *erreur*

`arbre_dec` ( { (c, []) } ) = L(c)

`arbre_dec` (tab) =

N ((),

`arbre_dec` { (c, m) | (c, 0.m) ∈ tab },

`arbre_dec` { (c, m) | (c, 1.m) ∈ tab } )

- [] est le mot vide,  $m_1 \cdot m_2$  concatène deux mots
- Lire les équations séquentiellement

# Construction d'une table de codage

...à partir d'un arbre de décodage.

`tab_cod` prend : mot  $m$  (initialement vide) et un arbre ; construit : table de codage.

`tab_cod (m, L(c)) = { (c, m) }`

`tab_cod (m, N((), g, d)) =`  
`tab_cod (m·0, g) ∪ tab_cod (m·1, d)`

⇒ “Arbre de décodage” et “Table de codage” sont des notions équivalentes.

*Exercices :*

- Représentez l'arbre `arb` de l'exemple
- Calculez `tab_cod [] arb`

# Inégalité de Kraft (1)

**Théorème :** Il existe un code préfixe binaire avec  $k$  codes  $u_1 \dots u_k$  si et seulement si

$$\sum_{i=1}^k 2^{-|u_i|} \leq 1$$

où  $|u_i|$  est la longueur du code  $u_i$ .

*Preuve :* devoir maison (voir TD).

*Exemple 1 :* Construisez un code préfixe pour 5 codes avec des longueurs  $|u_1| = 1$ ,  $|u_2| = 2$ ,  $|u_3| = 2$ ,  $|u_4| = 3$ ,  $|u_5| = 4$ .

On constate qu'un tel code n'existe pas parce que

$$\sum_{i=1}^5 2^{-|u_i|} = \frac{19}{16} > 1$$

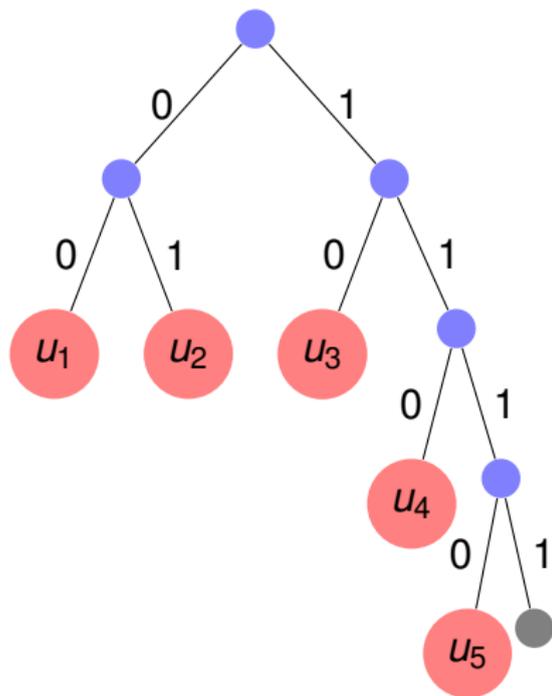
# Inégalité de Kraft (2)

*Exemple 2* : Construisez un code préfixe avec  $|u_1| = 2$ ,  $|u_2| = 2$ ,  $|u_3| = 2$ ,  $|u_4| = 3$ ,  $|u_5| = 4$ .

Un tel code existe parce que

$$\sum_{i=1}^5 2^{-|u_i|} = \frac{15}{16} \leq 1$$

*Idee* de construction de l'arbre :  
"Remplir" l'arbre à profondeur 2,  
ensuite 3, ensuite 4



# Résumé : Codage

- Hiérarchie de classes de codes : injectif, unique, préfixe
- Codes préfixes (aussi appelés *instantanés*) : Décodage possible “en temps réel”, sans attendre la fin du message.
- Considéré ici : Alphabet cible est  $\{0, 1\}$ . Extension vers alphabets de taille  $n > 2$  avec arbres  $n$ -aires possible.
- Inégalité de Kraft : (Im)possibilité de construire des codes avec certaines longueurs.

# Plan

- 1 Codage
- 2 La notion d'information selon Shannon et codage optimal**
- 3 Complexité de Kolmogorov et compression de données
- 4 Codes correcteurs et détecteurs d'erreurs

# Plan

- 2 La notion d'information selon Shannon et codage optimal
  - Information
  - Codage optimal

# Information (motivation) : un cas limite

*« Quid hic ? » inquam, « quis est qui complet aures meas tantus et tam dulcis sonus ? » « Hic est » inquit « ille qui [...] impulsu et motu ipsorum orbium efficitur. [...] Hoc sonitu opletæ aures hominum obsurduerunt ; nec est ullus hebetior sensus in vobis, sicut ubi Nilus ad illa quæ Catadupa nominantur præcipitat ex altissimis montibus, ea gens quæ illum locum adcolit propter magnitudinem sonitus sensu audiendi caret. »*

*Cicero : Somnium Scipionis*

- Scipio Aemilianus s'étonne d'un son inconnu entendu dans son rêve.
- Scipio Africanus répond que les oreilles remplies du son permanent sont incapables de l'apercevoir.

⇒ Un signal toujours présent n'est pas perçu comme informatif

# Information (motivation) : mesure de Hartley



## Première approximation :

Quelle est la capacité informationnelle d'émetteur  $X$  qui émet un

①  $s \in E_8 = \{000, \dots, 111\}$

②  $s \in E_{16} = \{0000, \dots, 1111\}$

**Observation :** rapport logarithmique entre

- taille  $|E|$  de l'ensemble des événements  $E$
- taille des mots pour décrire les éléments de  $E$

**Proposition (informelle) :**  $Info(X) := \log_2(|E|)$  bits

Voir :

R.V.L. Hartley : *Transmission of Information*, Bell Syst. Tech. J., 1928

# Information (motivation) : mesure de Shannon

Soit  $X$  une source qui émet des signaux

$$E_8 = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

Quelle est la complexité pour décrire

- un élément vert : “commence avec 0”  $\rightsquigarrow$  complexité 1
- un élément rouge : “commence avec 10”  $\rightsquigarrow$  complexité 2
- un élément bleu / noir : “est 110” / “est 111”  $\rightsquigarrow$  complexité 3

Observations :

- la fréquence d'occurrence d'éléments n'est pas uniforme
- élément plus fréquent  $\rightsquigarrow$  description plus compacte

**Proposition (informelle)** : pondérer complexité avec fréquence  
*Ici* : “la complexité moyenne” pour décrire un  $s \in E_8$  relatif aux événements désignés :

$$\text{Info}(X) = 0.5 * 1 + 0.25 * 2 + 0.125 * 3 + 0.125 * 3 = 1.75\text{bits}$$

# Types de signaux et sources



Émetteur : *source d'information* qui émet des *signaux aléatoires*.

Types de signaux :

- continu
- $\Rightarrow$  ici : discret :  $s_0, s_1, \dots$

Types de source :

- *Dépendance de valeurs antérieures* :
  - $\Rightarrow$  ici : sans mémoire
  - markovienne (signal  $s_{n+1}$  dépend de  $s_n$ )
- *Évolution au cours du temps* :
  - $\Rightarrow$  ici : stationnaire (pas d'évolution)
  - non-stationnaire

# Quelques notions de la théorie de probabilité (1)

## Espace de probabilité $(\Omega, \mathcal{A}, P)$ , où

- $\Omega$  est l'ensemble des résultats d'une expérience aléatoire  
*Ex. jeu de dés :  $\Omega = \{1, \dots, 6\}$*
- $\mathcal{A} \subseteq 2^\Omega$  est l'ensemble des événements  
*(Notation :  $2^\Omega$  ensemble des parties de  $\Omega$ )*  
*Ex. :  $A \in \mathcal{A}$  pour  $A = \{2, 4, 6\}$  est l'événement "nombre pair".*
- $P : \mathcal{A} \rightarrow [0 \dots 1]$  est une mesure de probabilité  
*Ex. :  $P(\{1\}) = \dots = P(\{6\}) = \frac{1}{6}$*

## Règles de bonne formation :

- $\Omega \in \mathcal{A}$ ,  $\mathcal{A}$  est stable sous complément, union (dénombrable), intersection (dénombrable), *donc* : si  $A \in \mathcal{A}$ , alors  $\bar{A} \in \mathcal{A}$  etc.
- $P(\Omega) = 1$  et  $\forall A \in \mathcal{A}. P(A) \geq 0$
- $P(\bigcup_{i \in I} A_i) = \sum_{i \in I} P(A_i)$  pour des  $A_i$  mutuellement disjoints et  $I$  dénombrable  
*Ex. :  $P(\{2, 4, 6\}) = P(\{2\}) + P(\{4\}) + P(\{6\}) = \frac{1}{2}$*

## Quelques notions de la théorie de probabilité (2)

**Variable aléatoire**  $X$  pour “mesurer” les résultats d'une expérience.

$X : \Omega \rightarrow S$ , où

- $S$  est un ensemble
- très souvent :  $S = \mathbb{R}$
- souvent :  $X$  est l'identité

*Notation :*

- pour var. aléatoire  $X$  et valeur  $x \in S$  :  

$$P_X(x) =_{\text{def}} P(X = x) =_{\text{def}} P(\{\omega \mid X(\omega) = x\})$$
- similaire :  $P(a \leq X \leq b) =_{\text{def}} P(\{\omega \mid a \leq X(\omega) \leq b\})$  etc.

*Exemple :*

- $\Omega$  : ensemble de cercles  $\{(c_i, r_i)\}$  caractérisés par coordonnée du centre  $c_i$  et rayon  $r_i$
- $X : \Omega \rightarrow \mathbb{R}$  avec  $X(c, r) = \pi r^2$
- $P(4.9 \leq X \leq 5.1)$  Proba que surface de cercle est entre 4.9 et 5.1

# Quelques notions de la théorie de probabilité (3)

**Espérance**  $E(X)$  d'une variable aléatoire  $X$  relative à fonction  $g$  :

$$E(g, X) =_{\text{def}} \sum_{i \in I} g(x_i) * P_X(x_i)$$

[*déf. habituelle* pour  $g = \lambda x$ .  $x$  est :  $E(X) = \sum_{i \in I} x_i * P_X(x_i)$ ]

*Exemple :*

Nombre de points (0 ... 5) obtenus dans un test fait par 12 personnes.

Approximation :  $P_X(x_i) = \frac{\#pers.(x_i)}{12}$

Points $x_i$	0	1	2	3	4	5
# personnes( $x_i$ )	1	2	4	3	1	1
$P_X(x_i)$	0.083	0.166	0.333	0.25	0.083	0.083

$E(X) =$

$$0 * 0.083 + 1 * 0.166 + 2 * 0.333 + 3 * 0.25 + 4 * 0.083 + 5 * 0.083 = 2.333$$

# Définition d'entropie (1)

**Entropie** (selon Shannon) :

$$H(X) =_{\text{def}} \sum_{i \in I} (-\log_2(P_X(x_i))) * P_X(x_i)$$

- *Problème* de calcul de log si  $P_X(x_i) = 0$   
 $\rightsquigarrow$  restriction à ensemble d'indices  $I$  avec  $P_X(x_i) \neq 0$
- *Entropie et espérance* :  $H(X) = E((\lambda x. -\log_2(P_X(x))), X)$
- Unité de l'entropie : *bit* ("binary digit")

*Exemple* : Lampe à 4 "couleurs" et probabilité que couleur soit visible :

Couleur $x_i$	vert	rouge	bleu	noir
$P_X(x_i)$	0.5	0.25	0.125	0.125

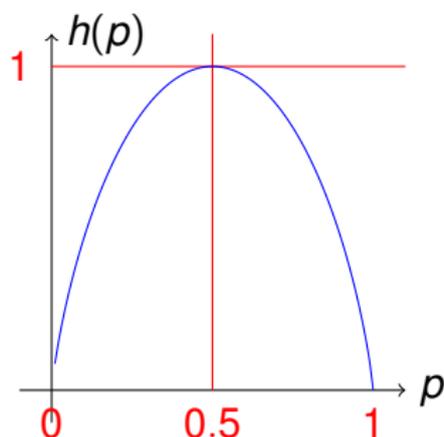
$$H(X) = (-\log_2(2^{-1})) * 0.5 + (-\log_2(2^{-2})) * 0.25 + (-\log_2(2^{-3})) * 0.125 + (-\log_2(2^{-3})) * 0.125 = 1.75\text{bits}$$

## Définition d'entropie (2)

*Illustration* : Soit  $X$  une var. aléatoire avec deux valeurs possibles  $x_1, x_2$  et

- $P_X(x_1) = p$
- donc :  $P_X(x_2) = 1 - p$

Alors,  $H(X) = -p \log_2(p) - (1 - p) \log_2(1 - p) =_{def} h(p)$



# Plan

- 2 La notion d'information selon Shannon et codage optimal
  - Information
  - Codage optimal

# Feuille de route

## Vu jusqu'à maintenant :

- Définition d'entropie d'une source d'information
- Codage homomorphe d'un alphabet avec un code préfixe
- ... (im)possibilité sous contraintes (théorème de Kraft)

## Dans cette section :

- Rapport entre entropie et longueur des mots d'un code : théorème de Shannon
- Construction effectif d'un codage optimal : algorithme de Huffman

## Cadre de travail :

- Source d'information stationnaire et sans mémoire
- Codage homomorphe : *un* caractère à la fois

# Arbres avec probabilités

[Discussion d'après : J. Massey : Applied Digital Information Theory]

- Dans un **arbre avec probabilités**, l'annotation d'un noeud est la somme des annotations de ses enfants.
- L'arbre est *complet* si la racine est annotée avec 1

*Exercice* : Pour un arbre  $a$ , définissez les fcts. calculant l'ensemble

- de tous les mots de code avec leur probabilité :

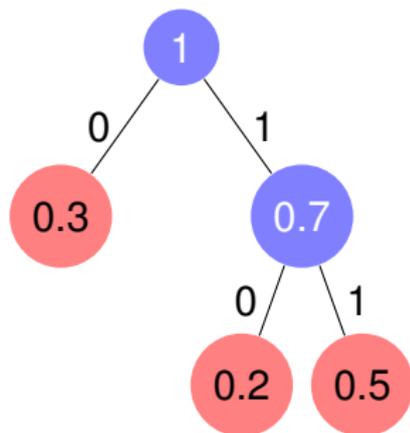
$$cp(a) = \{([], 1), (0, 0.3), (1, 0.7), (10, 0.2), (11, 0.5)\}$$

- des mots aboutissant à des feuilles, et proba :

$$cpf(a) = \{(0, 0.3), (10, 0.2), (11, 0.5)\}$$

- des noeuds intérieurs, et proba :

$$cpi(a) = \{([], 1), (1, 0.7)\}$$



# Lemme des longueurs des chemins

## Définitions :

- Longuer moyenne d'un ensemble (mot  $\times$  probabilité) :

$$Inm(E) = \sum_{(m,p) \in E} p * |m|$$

$$Exemple : Inm(cpf(a)) = 0.3 * 1 + 0.2 * 2 + 0.5 * 2 = 1.7$$

- Somme des probabilités d'un ensemble (mot  $\times$  probabilité) :

$$sp(E) = \sum_{(m,p) \in E} p$$

$$Exemple : sp(cpi(a)) = 1 + 0.7 = 1.7$$

**Lemme :** Dans un arbre avec probabilités, la longueur moyenne des mots aboutissant à des feuilles est égale à la somme des probabilités des noeuds intérieurs :

$$Inm(cpf(a)) = sp(cpi(a))$$

# Entropie d'un code

**Définition :** L'entropie d'un arbre est l'entropie de ses feuilles.

$$H(a) = - \sum_{(m,p) \in \text{cpf}(a)} p * \log_2 p$$

*Exemple :*

$$H(a) = -(0.3 * \log_2(0.3) + 0.2 * \log_2(0.2) + 0.5 * \log_2(0.5)) = 1.485$$

Sur calculette : calculer avec le logarithme naturel et diviser enfin par le logarithme naturel de 2.

**Définition :** L'entropie de branchement  $H_1(a)$  d'un arbre  $a$  qui n'est pas une feuille est l'entropie des fils directs de la racine, donc la somme ne se fait pas sur les feuilles mais les (deux) fils directs de la racine.

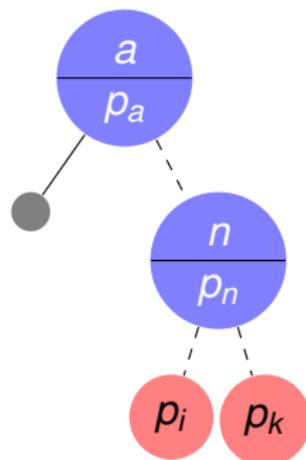
# Décomposition

**Théorème** (décomposition du calcul de l'entropie d'arbre) : Soit  $a$  un arbre complet. Alors :

$$H(a) = \sum_{n \in AI, \text{prob}(n) \neq 0} \text{prob}(n) * H_1\left(\frac{1}{\text{prob}(n)} * n\right)$$

- où  $AI$  est l'ensemble des sous-arbres intérieurs (non feuilles) de  $a$  (y compris  $a$  le cas échéant)
- $\text{prob}(N(p, a_0, a_1)) = p$
- la multiplication d'un arbre avec un nombre réel s'opère sur les probabilités (dans notre cas, pour assurer que l'arbre est complet à nouveau)

*Preuve* : voir p. 31 des notes de Massey.



# Théorème de Shannon, 1ère partie

**Théorème :** L'entropie d'un arbre complet de codage  $a$  est une *borne inférieure* de la longueur moyenne de son code :

$$H(a) \leq \text{Inm}(\text{cpf}(a))$$

*Preuve :* Selon le lemme de décomposition :

$$H(a) = \sum_{n \in AI, \text{prob}(n) \neq 0} \text{prob}(n) * H_1\left(\frac{1}{\text{prob}(n)} * n\right)$$

On peut montrer :  $H_1(a) \leq 1$  pour  $a$  complet

(Justification visuelle : courbe p. 59—ça vaut pour les arbres binaires)

Donc :

$$\begin{aligned} H(a) &\leq \sum_{n \in AI} \text{prob}(n) \\ &= \text{sp}(\text{cpi}(a)) && \text{déf. de } \text{sp} \text{ et } \text{cpi} \\ &= \text{Inm}(\text{cpf}(a)) && \text{Lemme des longueurs des chemins} \end{aligned}$$

# Théorème de Shannon, 2ème partie

**Théorème :** La *borne supérieure* de la longueur moyenne d'un code préfixe minimal pour une source d'information aléatoire  $U$  avec entropie  $H(U)$  est  $H(U) + 1$  :

$$\ln m(\text{cpf}(a)) < H(U) + 1$$

*Preuve :* On cherche code  $m_1 \dots m_k$  pour les caractères  $u_1 \dots u_k$  de  $U$ .

Choisissons  $|m_i| = \lceil -\log_2 P_U(u_i) \rceil$ , alors

$$-\log_2 P_U(u_i) \leq |m_i| < -\log_2 P_U(u_i) + 1$$

$$\text{On vérifie } \sum_{i=1}^k 2^{-|m_i|} \leq \sum_{i=1}^k 2^{\log_2 P_U(u_i)} = \sum_{i=1}^k P_U(u_i) = 1$$

Selon l'inégalité de Kraft, un tel code préfixe existe.

En plus :

$$\ln m(\text{cpf}(a)) = \sum_{i=1}^k |m_i| * P_U(u_i)$$

$$< \sum_{i=1}^k ((-\log_2 P_U(u_i)) + 1) * P_U(u_i)$$

$$= \sum_{i=1}^k (-\log_2 P_U(u_i)) * P_U(u_i) + \sum_{i=1}^k P_U(u_i) = H(U) + 1$$

# Algorithme de Huffman (1)

Comment construire *effectivement* un code optimal pour une source d'information aléatoire  $U$  avec alphabet  $\{u_1 \dots u_k\}$  ?

Démarche globale :

- 1 Déterminer la distribution de probabilité  $P_U(u_i)$  des caractères
  - par estimation (distribution des caractères en français, anglais, ...)
  - par calcul de la fréquence dans un texte donné

*Exemple* : un petit texte

$u_i$	␣	e	i	n	p	t	u	x
# occ	2	3	1	1	1	4	1	1
$P_U(u_i)$	$\frac{2}{14}$	$\frac{3}{14}$	...					

- 2 Construire l'arbre de codage
- 3 L'utiliser pour le codage du texte

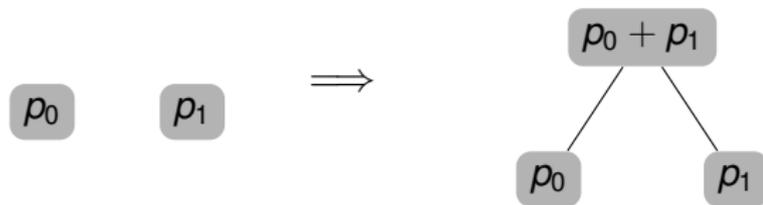
# Algorithme de Huffman (2)

**Structure de données** : Ensemble  $E$  d'arbres dont

- les feuilles contiennent : probabilité, caractère codé
- les noeuds intérieurs contiennent : probabilité cumulée

**Algorithme** pour construire un arbre de codage optimal  $a$  à partir de caractères  $u_1 \dots u_k$  et leurs probabilités associés  $p_1 \dots p_k$

- 1 **Initialisation** :  $E := \{L(p_i, u_i) \mid 1 \leq i \leq k\}$
- 2 Tant que  $E$  contient plus d'un élément :
  - 1 Sélectionner deux arbres  $a_0, a_1 \in E$  dont la probabilité est minimale
  - 2 Les remplacer par  $N(p, a_0, a_1)$ , où  $p$  est la somme des probas de  $a_0$  et  $a_1$



- 3 **Fin de boucle** :  $E = \{a\}$ . Renvoyer  $a$ .

# Algorithme de Huffman (3)

Exemple : Distribution de fréquence :

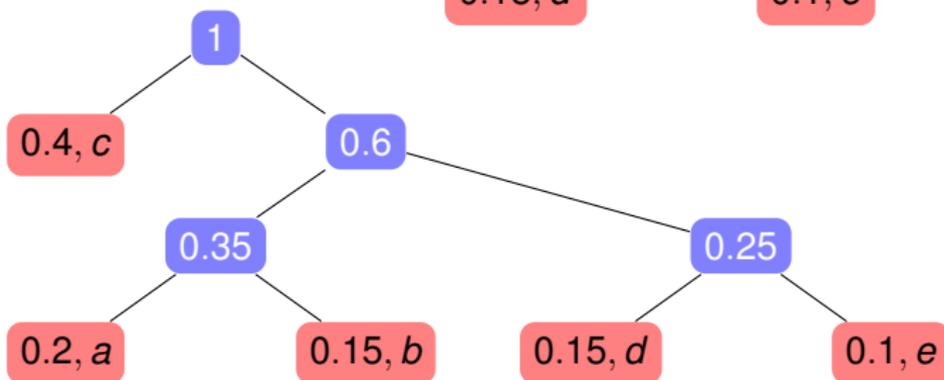
a	b	c	d	e
0.2	0.15	0.4	0.15	0.1

0.2, a   0.15, b   0.4, c   0.15, d   0.1, e

⇒

0.2, a   0.15, b   0.4, c   0.25  
 0.15, d   0.1, e

⇒\*



# Algorithme de Huffman (4)

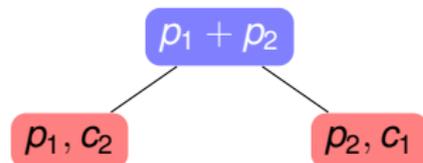
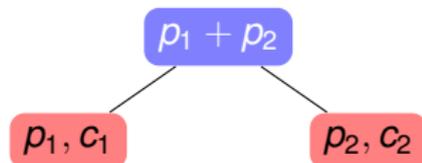
**Définition :** Code *optimal* pour une source d'information aléatoire  $U =$   
Code *minimal* parmi tous les codes pour  $U$

**Proposition :** L'algorithme de Huffman construit un code optimal pour  $U$

*Esquisse de preuve :* [Détails : Cover / Thomas : *Information Theory*]

Soit  $a_o$  un arbre de code optimal. Alors, il ne peut pas être meilleur que  $a_h$  construit par Huffman.

- $a_o$  n'a pas de noeud  $n$  avec un seul enfant  $e$   
*sinon* : fusion de  $n$  et  $e$  donne meilleur code
- $a_o$  code  $c_1 \mapsto m_1, c_2 \mapsto m_2$   
permuter pour obtenir code de  $a_h$  avec  $c_1 \mapsto m_2, c_2 \mapsto m_1$   
et même longueur moyenne du code



# Mise en garde

## Codage "optimal"

- pour l'hypothèse de travail :
  - source sans mémoire ; codage de caractères individuels
- ...mais non dans l'absolu

*Exemple :*

- ① Codage de caractères  $a$  ( $P(a) = \frac{1}{4}$ ) et  $b$  ( $P(b) = \frac{3}{4}$ )

Code Huffman :  $c(a) = 0$ ,  $c(b) = 1$ .

Longueur moyenne de codage par caractère : 1

- ② Codage de deux caractères consécutifs :

Code Huffman :  $c(aa) = 010$ ,  $c(ab) = 011$ ,  $c(ba) = 00$ ,  $c(bb) = 1$ .

Longueur moyenne de mot de code :

$$3 * \frac{1}{16} + 3 * \frac{3}{16} + 2 * \frac{3}{16} + 1 * \frac{9}{16} = 1.6875$$

Longueur moyenne de codage par caractère :  $1.6875/2$

⇒ Meilleurs résultats pour regroupements de caractères ...

# Plan

- 1 Codage
- 2 La notion d'information selon Shannon et codage optimal
- 3 Complexité de Kolmogorov et compression de données**
- 4 Codes correcteurs et détecteurs d'erreurs

# Plan

## 3 Complexité de Kolmogorov et compression de données

- Motivation et délimitation
- Complexité de Kolmogorov
- Algorithmes de Compression

# Critique : Information selon Shannon (1)

## Rappel : Entropie :

- Notion *probabiliste* : dépend de la probabilité des caractères
- ... laquelle est typiquement estimée à partir d'un corpus large (*exemple* : tous les textes en français de la bibliothèque nationale)

## Codage optimal :

- Sur la base d'une connaissance *a priori* de la distribution de probabilité  
donc : de l'ensemble de *tous* les messages
- Indépendant d'un message individuel

# Critique : Information selon Shannon (2)

## Défauts et insuffissances de la notion de Shannon :

- On s'intéresse souvent à la compression d'un seul message / fichier donné (et non à tous les messages concevables)
- On ne connaît pas la distribution de probabilité
- Il y a d'autres types de redondance
- *Exemple* : Compression d'un fichier écrit en Python
  - Distribution de caractères représentative pour un texte en français / en anglais ?
  - Peu de variations syntaxiques (langage artificiel)
  - Répétition de mots-clés (`while`, `if`) et noms de variables
- *Exemple* : Compression d'une image
  - Des régions de l'image ont la même couleur  
"le rectangle entre les coordonnées (15, 42) et (37, 98) est bleu"









# Survol de ce chapitre

- La notion de complexité de Kolmogorov :  
“La plus courte description pour une chaîne de caractères donnée”
  - *But* : Une autre vue sur la notion de contenu informationnel que l'entropie
  - *Chemin faisant, une découverte* : un problème insoluble !  
Impossibilité de calculer la plus courte description
- Une application pratique : l'algorithme LZW  
compression d'un texte à l'aide d'un dictionnaire  
*But* : Les rouages des algorithmes de la famille `zip`

# Plan

## 3 Complexité de Kolmogorov et compression de données

- Motivation et délimitation
- **Complexité de Kolmogorov**
- Algorithmes de Compression



# Le paradoxe de Berry (1)

Définissons le nombre  $n$  par :

*le nombre le plus court non descriptible en moins de treize mots*

(NB : la description comporte douze mots)

Trouvez le nombre  $n$

Il est descriptible avec combien de mots ?

# Le paradoxe de Berry (2)

Le langage mathématique n'est pas exempt de ces problèmes et permet d'énoncer des descriptions insensées.

Quelles définitions acceptez vous ?

- Ensembles :  $E_1 = \{E \mid E \notin E\}$   
(est-ce que  $E_1 \in E_1$  ou  $E_1 \notin E_1$  ?)
- Ensembles :  $E_2 = \{n \in \mathbf{N} \mid n \bmod 3 = 0\}$
- Récursion : La fonction  $f$  telle que pour tout  $x$ ,  $f(x) = f(x) + 1$
- Récursion : La fonction  $f : \mathbf{N} \rightarrow \mathbf{N}$  telle que :
  - $f(0) = 0$
  - $f(n) = n + f(n - 1)$  pour  $n > 0$

Conclusion :

- il faut préciser la notion de “description”
- et préférer une notion *constructive*

# Descriptions constructives

La **description verbale** “ $n$  fois 01”  
remplacé par un **programme** (avec paramètre  $n$ ) dans un langage de programmation fixe :

```
for i in [1 .. n]
  print("01");
```

*Avantage :*

- un langage de programmation a une sémantique précise
- évite les ambiguïtés du langage naturel

*Quel langage précisément ?*

- Imaginez-vous Python ...
- mais ça pourrait être C, Java  
sans impact essentiel sur le résultat (voir TD)
- *Important* : le langage est *fixe*

# Programmes et leur code binaire (1)

Nous considérons désormais les programmes dans un **format binaire**

*Exemple :*

- Format textuel :

```
Programme  $p = \text{for } i \text{ in } [1 .. n] \text{ print}("01");$ 
```

Argument  $n = 20$

- Format binaire :

```
 $p = 1001101101010111$ 
```

(phantaisiste - en ASCII, UTF-8, ...) *Important* : format fixe

```
 $n = 10100$ 
```

*NB* : Quelques programmes produisent des séquences infinies  
(sans importance pour la discussion suivante) :

```
while true  
    print("1");
```



# Programmes et leur code binaire (3)

Deuxième avantage : On peut **trier les fonctions selon leur code**

Programme (texte)	Programme (binaire)	Progr. indexé (décimal)
<code>print("0");</code>	10001	$p_{17}$
<code>print("1");</code>	10101	$p_{21}$
...		
<code>for i in ...</code>	1001101101010111	$p_{39767}$
...		

Beaucoup d'indices correspondent à des programmes mal formés.

Ex. : `while print ();`

On ne les liste pas.

# Fonctions calculables

## Nous avons vu :

- Des programmes représentés comme séquences  $\{0, 1\}^*$
- qui prennent des entrées  $\{0, 1\}^*$
- et produisent des sorties  $\{0, 1\}^*$

Donc : un programme est une fonction  $\{0, 1\}^* \rightarrow \{0, 1\}^*$

	$\langle p_i, 0 \rangle$	$\langle p_i, 1 \rangle$	$\langle p_i, 10 \rangle$	$\langle p_i, 11 \rangle$	...
$p_{17}$	0	0	0	0	...
$p_{21}$	1	1	1	1	...
$p_{39767}$	[ ]	01	0101	010101	...

Une fonction est dite **calculable** s'il y a un programme qui la représente.

# Fonctions non calculables (1)

*Question* : est-ce que toute fonction  $\{0, 1\}^* \rightarrow \{0, 1\}^*$  est calculable ?

Supposons que oui.

Soit  $p_0, p_1, p_2 \dots$  l'énumération des programmes

et  $s_0, s_1, s_2 \in \{0, 1\}^*$  l'énumération des séquences

Dérivons une contradiction par **diagonalisation** :

	$\langle p_i, s_0 \rangle$	$\langle p_i, s_1 \rangle$	$\langle p_i, s_2 \rangle$	$\langle p_i, s_3 \rangle$	...
$p_0$	1 0	0	0	0	...
$p_1$	1	0 1	1	1	...
$p_2$	[ ]	01	1101 0101	010101	...
...					

La fonction qui diffère sur la diagonale n'est pas dans la liste des programmes

## Fonctions non calculables (2)

*De manière plus formelle :*

Définissons, pour une séquence  $s$ , la fonction “rendre différent”,  $\bar{s}$ , par  $\overline{0} = 1$  et  $\overline{0s'} = 1s'$  et  $\overline{1s'} = 0s'$

*Constat :* Pour tout  $s : \bar{s} \neq s$

Définissons la fonction  $nc$  (“non calculable”) par :

$$nc(s_i) = \overline{\langle p@i, s_i \rangle}$$

Supposons que  $nc$  est représenté par le programme à la position  $k$  :  
 $nc = p@k$ .

$$\text{Alors, } nc(s_k) = \overline{\langle p@k, s_k \rangle} \neq \langle p@k, s_k \rangle = p@k(s_k)$$

Contradiction.

Une fonction moins artificielle que  $nc$ ? Dans quelques instants ...

# Complexité de Kolmogorov : Définition précise

## Définissons

- le **décodeur de Kolmogorov**  $d_K$  par  
 $d_K(\langle p, n \rangle) = \text{le résultat de } p(n)$   
(indéfini si  $p$  n'est pas un programme valide)
- le **code de Kolmogorov**  $c_K$  par  
 $c_K(x) = \text{le plus court et plus petit (comme nombre binaire) } y \text{ tq.}$   
 $d_K(y) = x$
- la **complexité de Kolmogorov**  $K(x) = |c_K(x)|$

*Lemme* : Pour tout  $n$ , il existe  $x$  avec  $|x| = n$  tq.  $K(x) \geq n$

*Preuve* : voir TD

# Complexité de Kolmogorov non calculable (1)

Supposons  $K$  est calculable. Nous pouvons définir le programme (bien défini à cause du lemme) :

```
sequenceComplexe(n) =
  for s with |s| = n
    if K(s) ≥ n:
      print(s); return;
```

Soit  $p$  le programme `sequenceComplexe` avec sous-programme  $K$ .  
Choisissons  $m$  avec  $m > |\langle p, m \rangle|$  (voir TD).

Argument informel :

- Exécutons `sequenceComplexe(m)`, nous obtenons  $s$  avec  $K(s) \geq m$
- $s$  ne peut donc pas être produit par un programme plus court que  $m$
- Mais la configuration  $\langle p, m \rangle$  est plus courte que  $m$
- Contradiction !

# Complexité de Kolmogorov non calculable (2)

*Plus formellement :*

- $d_K(\langle p, m \rangle) = \text{sequenceComplexe}(m) = s$  avec  $K(s) \geq m > |\langle p, m \rangle|$ .
- Par contre :  $|\langle p, m \rangle| \geq |c_K(s)| = K(s)$
- Contradiction, donc :  **$K(x)$  n'est pas calculable**

(Vous avez un effet de déjà-vu ? Comparez avec le paradoxe de Berry)

# Résumé

## But :

- Développer un autre concept d’“information” que l’entropie
- Notion de compression optimale pour message individuel (concept non probabiliste)

## Difficultés :

- Notion informelle trop imprécise  $\rightsquigarrow$  paradoxes
- Définition formelle : complexité de Kolmogorov

## Résultat : Complexité de Kolmogorov non calculable :

- Impossibilité d’une solution algorithmique
  - Impossibilité forte (mathématique) . . .
  - et non au sens conventionnel (pas de temps, manque d’envie, trop stupide . . .)
- [ Publicité : cours “Calculabilité” du L3 ]

## Pourtant : regardons des algorithmes sans garantie d’optimalité !

# Plan

## 3 Complexité de Kolmogorov et compression de données

- Motivation et délimitation
- Complexité de Kolmogorov
- Algorithmes de Compression

# Pour situer le contexte

**Entropie** : Notion d'information d'une source

- qui émet des signaux / caractères avec une certaine probabilité
- où les signaux se produisent de manière indépendante (“sans mémoire”)

**Codage de Huffman** : Optimal pour une telle source d'information

**Une hypothèse réaliste ?**

- *Oui*, si vous n'avez aucune connaissance supplémentaire
- *Non* pour une source d'informations plus structurées

*Exemples :*

- textes en langage naturel
- programmes
- images avec beaucoup de répétitions

**Ici** : Algorithmes de compression à base de dictionnaires

# Compression à base de dictionnaires – Idée

Un texte à transmettre, par exemple :

*Une compression à base d'un dictionnaire peut être plus efficace qu'une compression à base d'entropie.*

Un dictionnaire partagé entre l'émetteur et le récepteur, p. ex. le Littré :

Posit. du mot	mot	définition
1	<b>a</b>	Voyelle et première lettre de l'alphabet.
2	<b>a</b>	3e pers. sing. du verbe <i>avoir</i> .
3	<b>à</b>	(préposition)
...		
5.233	<b>compression</b>	L'état qui résulte de la compression.
...		
60.582/3	<b>un, une</b>	Adjectif numéral

Le texte codé :

[60.583, 5.233, 3, 1.220, 7.588, 60.582, 8.122, ...]

# Méthodes statiques et adaptatives

## Méthodes statiques : (exemple précédent)

- Dictionnaire partagé entre émetteur et récepteur.
- Dans le texte codé, chaque mot du texte source est remplacé par sa position dans le dictionnaire.
- Fréquence de mise à jour et transmission du dictionnaire est négligeable.
- $\rightsquigarrow$  moyennement adapté au langage naturel (voir TD).
- $\rightsquigarrow$  inadapté aux applications informatiques.

## Méthodes adaptatives :

- Sans dictionnaire partagé, ou dictionnaire partagé minimaliste.
- Lors du codage, on construit un dictionnaire et le texte compressé.
- Seulement le texte compressé est transmis.
- $\rightsquigarrow$  la base des algorithmes actuels de codage.

# Compression à base de dictionnaires – Histoire (1)

**LZ77** : Jacob Ziv et Abraham Lempel : *A Universal Algorithm for Sequential Data Compression* (1977)

Algorithme à base d'une "fenêtre" qui glisse sur un texte.

Dictionnaire : parties de texte de la fenêtre

*Exemple* d'un texte de programme à coder :

- Texte déjà lu et codé jusqu'au dernier caractère de la fenêtre :

```
begin for(i=0; i<MAX-1; i++) for(j=i+1; j<MAX; j++)
```

- Dans le texte à coder, on reconnaît un sous-texte de la fenêtre :

```
begin for(i=0; i<MAX-1; i++) for(j=i+1; j<MAX; j++)
```

- On code `<MAX` par le triplet (11, 4, ';') : sa position relative au début de la fenêtre ; sa longueur ; le caractère suivant

- On avance la fenêtre et continue à coder :

```
begin for(i=0; i<MAX-1; i++) for(j=i+1; j<MAX; j++)
```

## Compression à base de dictionnaires – Histoire (2)

**LZ78** : Jacob Ziv et Abraham Lempel : *Compression of Individual Sequences via Variable-Rate Coding* (1978)

- Plus de fenêtre glissante, mais . . .
- un dictionnaire de chaînes de caractères rencontrées avant.

**LZW** : Terry Welch : *A Technique for High-Performance Data Compression* (1984)

- Extension de l'algorithme LZ78
- Différence essentielle : dictionnaire initialisé avec tous les caractères de l'alphabet.

# Compression à base de dictionnaires – Histoire (3)

## Compression de textes :

- `compress` : utilitaire du OS Unix qui utilisait LZW
- *Problème* : LZW était breveté par Sperry / Unisys corporation
- `gzip` : programme de compression alternatif (et libre), combinaison de LZ77 et codage de Huffman.

## Compression d'images :

- GIF (Graphics Interchange Format) basé sur LZW souffre du problème du brevet de LZW
- Alternative : PNG / PING (“PING is not GIF”)  
compression : LZ77 et autres

## Une multitude de programmes combinant différentes approches :

- 7-Zip : basé sur algorithme Lempel-Ziv-Markov, combinaison de LZ77 et chaînes de Markov (codage d'entropie)
- `bzip` : transformation de Burrows-Wheeler ; transformation Move-to-Front ; codage de Huffman
- ...

# LZW en détail

## Structure de données : le dictionnaire

- *en principe* : un tableau de chaînes de caractères

'a'	'b'	'c'	'ab'	'ba'
-----	-----	-----	------	------

- *ici, vue plus convenable* : association (chaîne  $\mapsto$  position)  
donc : dictionnaire au sens de Python :  
{ 'a' : 0, 'b' : 1, 'c' : 2, 'ab' : 3, 'ba' : 4 }

## Initialisation du dictionnaire :

- Initialisation avec l'ensemble des caractères de l'alphabet
- ... dans un ordre convenu par l'émetteur et le récepteur
- Ex. : Alphabet {a, b, c}, dict. : { 'a' : 0, 'b' : 1, 'c' : 2 }
- Ex. : Alphabet ASCII, dict. : caractère  $\mapsto$  code ASCII

# LZW : compression (1)

**Entrée :** Une chaîne de caractères `str`

**Sortie :** Une liste `compr` de positions dans le dictionnaire

**Algorithme :** construit en même temps

- la sortie `compr`
- le dictionnaire `dict`

1 Initialis. de `dict`, `compr = []`, mot partiel `m = ""` (chaîne vide)

2 Boucle : Pour tout caractère `c` de `str` : (\*)

- si `m + c` est dans `dict`, étendre `m` avec `c`
- si `m + c` n'est pas dans `dict` :
  - Ajouter `dict[m]` à `compr`
  - Ajouter `m + c` à la dernière position de `dict`
  - Mettre `m=c`

3 Pour finir, rajouter `dict[m]` à `compr`

# LZW : compression (2)

*Exemple pour l'alphabet  $\{a, b, c\}$*

- Entrée : ' abacababac '
- Dictionnaire initial : {'a' : 0, 'b' : 1, 'c' : 2}
- Dictionnaire final :  
 {'a' : 0, 'b' : 1, 'c' : 2, 'ab' : 3, 'ba' : 4,  
 'ac' : 5, 'ca' : 6, 'aba' : 7, 'abac' : 8}
- Résultat compr : [0, 1, 0, 2, 3, 7, 2]

*Reconstruction :*

0	1	0	2	3	7	2
a	b	a	c	ab	aba	c

# LZW : compression (3)

*Zoom* sur la phase de construction, à la position (\*) de l'algorithme :

- `dict = {'a': 0, 'b': 1, 'c': 2}`  
  `}`
- `compr = []`
- `str = abacababac`, `m=""`, `c=a=str[0]`

# LZW : compression (3)

*Zoom* sur la phase de construction, à la position (\*) de l'algorithme :

- `dict = {'a': 0, 'b': 1, 'c': 2}`  
`}`
- `compr = []`
- `str = abacababac`, `m=a`, `c=b=``str[1]`

# LZW : compression (3)

*Zoom* sur la phase de construction, à la position (\*) de l'algorithme :

- `dict = {'a': 0, 'b': 1, 'c': 2, 'ab': 3}`  
`}`
- `compr = [0]`
- `str = abacababac`, `m=b`, `c=a=``str[2]`

# LZW : compression (3)

*Zoom* sur la phase de construction, à la position (\*) de l'algorithme :

- `dict = {'a': 0, 'b': 1, 'c': 2, 'ab': 3, 'ba': 4}`  
  }
- `compr = [0, 1]`
- `str = abacababac`, `m=a`, `c=c=str[3]`

# LZW : compression (3)

*Zoom* sur la phase de construction, à la position (\*) de l'algorithme :

- `dict = {'a': 0, 'b': 1, 'c': 2, 'ab': 3, 'ba': 4, 'ac': 5}`
- `compr = [0, 1, 0]`
- `str = abacababac`, `m=c`, `c=a=``str[4]`

# LZW : compression (3)

*Zoom* sur la phase de construction, à la position (\*) de l'algorithme :

- `dict = {'a': 0, 'b': 1, 'c': 2, 'ab': 3, 'ba': 4, 'ac': 5, 'ca': 6 }`
- `compr = [0, 1, 0, 2]`
- `str = abacababac`, `m=a`, `c=b=``str[5]`

# LZW : compression (3)

*Zoom* sur la phase de construction, à la position (\*) de l'algorithme :

- `dict = {'a': 0, 'b': 1, 'c': 2, 'ab': 3, 'ba': 4, 'ac': 5, 'ca': 6 }`
- `compr = [0, 1, 0, 2]`
- `str = abacababac`, `m=ab`, `c=a=str[6]`

# LZW : compression (3)

*Zoom* sur la phase de construction, à la position (\*) de l'algorithme :

- `dict = {'a': 0, 'b': 1, 'c': 2, 'ab': 3, 'ba': 4, 'ac': 5, 'ca': 6, 'aba': 7 }`
- `compr = [0, 1, 0, 2, 3]`
- `str = abacababc, m=a, c=b=str[7]`

# LZW : compression (3)

*Zoom* sur la phase de construction, à la position (\*) de l'algorithme :

- `dict = {'a': 0, 'b': 1, 'c': 2, 'ab': 3, 'ba': 4, 'ac': 5, 'ca': 6, 'aba': 7 }`
- `compr = [0, 1, 0, 2, 3]`
- `str = abacababac`, `m=ab`, `c=a=str[8]`

# LZW : compression (3)

*Zoom* sur la phase de construction, à la position (\*) de l'algorithme :

- `dict = { 'a': 0, 'b': 1, 'c': 2, 'ab': 3, 'ba': 4, 'ac': 5, 'ca': 6, 'aba': 7 }`
- `compr = [0, 1, 0, 2, 3]`
- `str = abacababac`, `m=aba`, `c=c=str[9]`

# LZW : compression (3)

Fin de l'algorithme :

- `dict = {'a': 0, 'b': 1, 'c': 2, 'ab': 3, 'ba': 4, 'ac': 5, 'ca': 6, 'aba': 7, 'abac': 8 }`
- `compr = [0, 1, 0, 2, 3, 7]`
- `str = abacababac, m=c`

# LZW : compression (3)

Fin de l'algorithme :

- dict= {'a': 0, 'b': 1, 'c': 2, 'ab': 3, 'ba': 4, 'ac': 5, 'ca': 6, 'aba': 7, 'abac': 8 }
- compr = [0, 1, 0, 2, 3, 7, 2]
- str = `abacababac`

# LZW : Correction de la compression (1)

**Notion de correction** : Le dictionnaire et le code comprimé permettent de reconstruire la chaîne originale.

*Notation* : Le “dictionnaire inversé”  $\overline{dict}$  de  $dict$  est le dictionnaire qui échange clés contre valeurs, *par ex.* :

$dict = \{ 'a' : 0, 'b' : 1, 'c' : 2, 'ab' : 3, 'ba' : 4 \},$

$\overline{dict} = \{ 0 : 'a', 1 : 'b', 2 : 'c', 3 : 'ab', 4 : 'ba' \}$

Soit  $lc = len(compr)$  la longueur du code comprimé.

**Un invariant** : En parcourant la chaîne d'entrée  $str$ , pour toute position  $i \in \{0 \dots len(str)\}$  :

$$\overline{dict}[compr[0]] + \overline{dict}[compr[1]] + \dots + \overline{dict}[compr[lc - 1]] + m = str[0 : i - 1]$$

# LZW : Correction de la compression (2)

**Exemple** (et ce n'est pas une preuve !):

- $\text{dict} = \{ 'a': 0, 'b': 1, 'c': 2, 'ab': 3, 'ba': 4, 'ac': 5, 'ca': 6, 'aba': 7 \}$
- $\text{compr} = [0, 1, 0, 2, 3]$
- $\text{str} = \text{abacababac}$ ,  $m = a$ ,  $c = b = \text{str}[8]$
- **Donc**:  $\text{str}[0 : 7] = \text{abacaba} =$   
 $\overline{\text{dict}}[0] + \overline{\text{dict}}[1] + \overline{\text{dict}}[0] + \overline{\text{dict}}[2] + \overline{\text{dict}}[3] + m$

# LZW : Correction de la compression (3)

## Preuve :

- ① Montrer que l'invariant est vrai après la phase d'initialisation
- ② Montrer que chaque exécution de la boucle maintient l'invariant ; distinguez entre :
  - ① le cas où  $m + c$  est dans `dict` (modification de `m`)
  - ② le cas où  $m + c$  n'est pas dans `dict` (extension de `dict`)
- ③ Considérez la phase finale de la compression (rajout du dernier `m` à `dict`)

## Complétez les détails !

**Conclusion :** Étant donnée la chaîne d'entrée `str`, l'algorithme de compression construit un dictionnaire `dict` et une séquence de codes `compr` tels que :

$$\overline{dict[compr[0]]} + \overline{dict[compr[1]]} + \cdots + \overline{dict[compr[lc - 1]]} = str$$

où  $lc = len(compr)$

# LZW : décompression (1)

**Première idée :** Selon le théorème de correction de compression, on reconstruit `str` à l'aide de `dict` et `compr`.

*Problème :*

- Le décodeur connaît uniquement le code comprimé `compr`,
- mais le dictionnaire n'est pas transmis

**Deuxième idée :**

- On reconstruit le dictionnaire inversé  $dinv = \overline{dict}$  lors de la décompression
- ... et on l'utilise pour décoder.

# LZW : décompression (2)

Lors de la décompression de `compr = [0, 1, 0, 2, 3, 7, 2]`

- `dinv = {0: 'a', 1: 'b', 2: 'c', 3: 'ab', 4: 'ba', 5: 'ac'}`
- `str = abac`

Code antérieur : 2: 'c'

Code actuel : 3: 'ab'

- `dinv = {0: 'a', 1: 'b', 2: 'c', 3: 'ab', 4: 'ba', 5: 'ac', 6: 'ca'}`
- `str = abacab`

# LZW : décompression (3)

**Entrée** : Une liste `compr` de codes

**Sortie** : Une chaîne de caractères `str`

**Algorithme (première version)** : construit en même temps :

- la sortie `str`
- le dictionnaire `dinv`

Utilise variables `m_ant` (mot antérieur) et `m_act` (mot actuel)

- 1 Initialiser `dinv, m_act = dinv[compr[0]], str= m_act`
- 2 Pour tout code `k` **in** `compr[1:]` :  
    `m_ant = m_act`  
    `m_act = dinv[k]`  
    Ajouter `m_ant + m_act[0]` à `dinv`  
    Concaténer `m_act` à `str`

# LZW : décompression (4)

**Question :** Est-ce que  $d_{inv}[k]$  est toujours bien défini ? *Réponse :* Malheureusement non !

Lors de la décompression de  $compr = [0, 1, 0, 2, 3, 7, 2]$

- $d_{inv} = \{0: 'a', 1: 'b', 2: 'c', 3: 'ab', 4: 'ba', 5: 'ac', 6: 'ca'\}$
- $str = abacab$

Code antérieur : 3: 'ab'

Code actuel : 7: ???

**Analyse :** Dans le code original :

$str = abacababac$

la fonction de compression a rajouté  $aba: 7$  à  $dict$   
et directement émis 7.

# LZW : décompression (5)

## Algorithme (deuxième version) :

- 1 Initialisations : comme avant
- 2 Pour tout code  $k$  **in**  $\text{compr}[1:]$  :
  - $m\_ant = m\_act$
  - **si**  $k$  **in**  $\text{dinv} : m\_act = \text{dinv}[k]$   
Rajouter  $m\_ant + m\_act[0]$  à  $\text{dinv}$   
(comme avant)
  - **sinon** :  $m\_act = m\_ant + m\_ant[0]$   
Rajouter  $m\_act$  à  $\text{dinv}$

Concaténer  $m\_act$  à  $\text{str}$

# LZW : décompression (6)

*L'exemple continué :*

- $d_{inv} = \{0: 'a', 1: 'b', 2: 'c', 3: 'ab', 4: 'ba', 5: 'ac', 6: 'ca'\}$
- $str = abacab$

Code antérieur : 3: 'ab'

Code actuel : 7: ???

Calculer :  $m_{act} = 'ab' + 'ab'[0] = 'aba'$

- $d_{inv} = \{0: 'a', 1: 'b', 2: 'c', 3: 'ab', 4: 'ba', 5: 'ac', 6: 'ca', 7: 'aba'\}$
- $str = abacababa$

# Résumé

La famille des algorithmes LZ est

- basée sur des dictionnaires :  
références vers des séquences de texte vues précédemment
- compatible / peut être combinée avec des méthodes probabilistes : codage de Huffman / chaînes de Markov
- implantée dans la plupart des programmes de compressions actuels

# Plan

- 1 Codage
- 2 La notion d'information selon Shannon et codage optimal
- 3 Complexité de Kolmogorov et compression de données
- 4 Codes correcteurs et détecteurs d'erreurs**

# Plan

## 4 Codes correcteurs et détecteurs d'erreurs

- **Motivation**
- Détection d'erreurs : Notions et exemples de base
- Détection d'erreurs : CRC

# Erreurs de transmission et de stockage (1)

Erreurs de **transmission** sur un canal bruité à cause de :

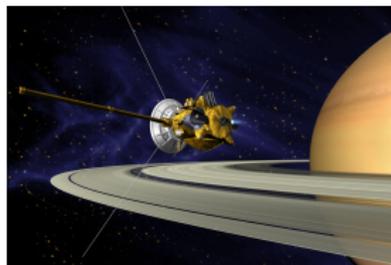
- influence d'ondes électromagnétiques
- signal trop faible

Erreurs de **stockage** à cause de :

- vieillissement du matériel
- contact avec des substances magnétiques (disques durs)
- dégradation physiques (CD/DVD)
- trop grand nombre de cycles d'écriture (mémoire flash, USB)

# Erreurs de transmission et de stockage (2)

*Exemple : la sonde Cassini*



Rayonnement cosmique causant des erreurs de mémoire de Cassini :

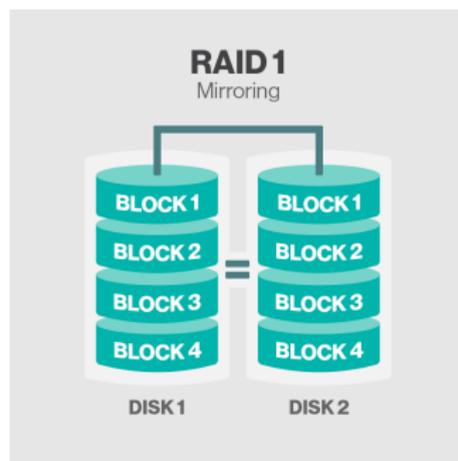
*The level is nearly constant at about 280 errors per day.*

Mais :

*On November 1997, the number of errors increased by about a factor of four . . . due to the coincidence in time of a small solar proton event.*

# Redondance (1)

Redondance par multiplication des ressources :  
**Mémoire**, par exemple l'architecture RAID  
(redundant array of independent disks)

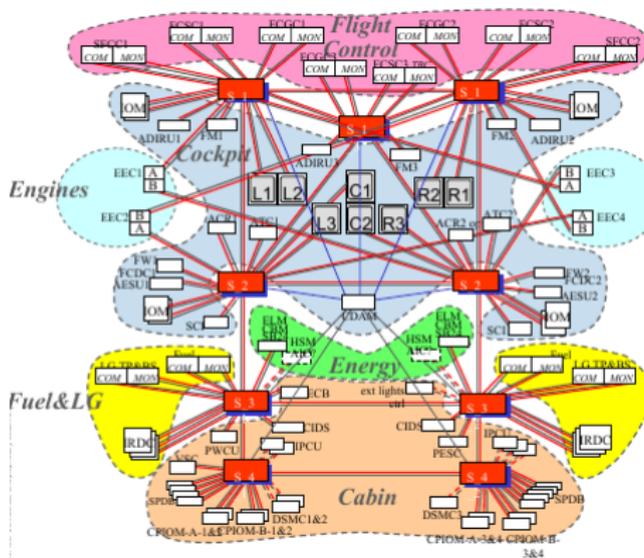


source : <http://searchstorage.techtarget.com/definition/RAID>

*Est-il utile de se limiter à deux disques ?*

# Redondance (2)

Redondance par multiplication des ressources :  
**Réseau de communication**, par exemple dans un A380



source : [http://www.artist-embedded.org/docs/Events/2007/IMA/Slides/ARTIST2\\_IMA\\_Itier.pdf](http://www.artist-embedded.org/docs/Events/2007/IMA/Slides/ARTIST2_IMA_Itier.pdf)

# Détection et correction d'erreurs

## Détection d'erreurs

- *But* : détecter l'occurrence d'une erreur, sans vouloir la corriger en même temps
- *Correction* : en utilisant un autre mécanisme de redondance :
  - accès à un serveur *backup*
  - retransmission des données (sur le même canal ou un autre)
- *Ici* : Contrôle de Redondance Cyclique

## Correction d'erreurs

- *But* : détecter et corriger des erreurs en même temps
- ... permet de se passer d'autres mécanismes de redondance
- *Ici* : Codes de Hamming

# Nécessité d'une redondance "intelligente"

## Principes :

- Plus de redondance permet de détecter / corriger plus d'erreurs
- ... une redondance "brute" est souvent inutile
- Il n'y a pas de détection / correction parfaite

## Illustration : Explorez les scénarios suivants (pour $n = 1, 2, 3$ ) :

- Envoi d'un message sur  $n$  canaux parallèles et indépendants.
- La probabilité d'erreur de chaque canal est 0.1.
- Pour  $n > 1$ , en cas de désaccord des messages reçus : arbitrage majoritaire

## Discutez :

- Possibilité de détection / correction des erreurs ?
- Probabilité de transmission correcte / de correction correcte / de correction erronée / de situation irrésoluble ?

# Plan

## 4 Codes correcteurs et détecteurs d'erreurs

- Motivation
- **Détection d'erreurs : Notions et exemples de base**
- Détection d'erreurs : CRC

## Exemple : clé RIB (1)

Un numéro de RIB (*Relevé d'Identité Bancaire*)  $R$  est composé de :

- $B$  : banque (5 chiffres)
- $G$  : guichet (5 chiffres)
- $N$  : numéro de compte (11 lettres alphanumériques)
- $C$  : clé (2 chiffres)

La clé  $C$  est déterminée par  $B$ ,  $G$ ,  $N$  :

- Calculer  $S(N)$  par une traduction, caractère par caractère :

$A \rightarrow 1, B \rightarrow 2, \dots, I \rightarrow 9, J \rightarrow 1, \dots, S \rightarrow 2(!!!) \dots Z \rightarrow 9$

(voir tableau sur feuille de TD),  $1 \rightarrow 1 \dots 9 \rightarrow 9$

*Exemple* :  $S(TINF17 \dots) = 395617 \dots$

- Critère de correction de  $C$  : la concaténation des nombres  $B$ ,  $G$ ,  $S(N)$ ,  $C$  forme un nombre à 23 chiffres divisible par 97

## Exemple : clé RIB (2)

**Exemple :** *Vérification d'un RIB*

Soit  $B = 36187$ ,  $G = 04329$ ,  $N = A35ACDC94IR$ ,  $C = 25$

Calculons  $S(N) = 13513439499$

Concaténation :  $R = 36187043291351343949925$

Vérification :  $R \bmod 97 = 0$

**Note :**

- Méthode très similaire pour la clé d'un IBAN (*International Bank Account Number*)
- Pareil : Clés pour le numéro de sécurité sociale, ISBN (identifiant unique de livres), code-barres (identification de produits etc.)

# Exemple : Bit de parité (1)

## Principe :

- Pour représentation *binaire* des données.
- On rajoute 1 bit redondant (le *bit de parité*) pour transmettre un message de  $n$  bits.

## Variantes :

- *Parité paire* : La somme des bits (incl. bit de parité) modulo 2 est 0
- *Parité impaire* : La somme des bits (incl. bit de parité) modulo 2 est 1

## Typiquement / historiquement :

- $n = 7$  (longueur du code ASCII)
- Avec le bit de parité, on code un caractère ASCII en un octet (8 bits)

## Exemple : Bit de parité (2)

**Exemples** (ici : parité paire ; bit de parité rajouté à la fin)

*Codage d'un message*

Message	Somme des bits	Bit de parité	Message codé
0110110	4	0	0110110 <b>0</b>
1101011	5	1	1101011 <b>1</b>

*Décodage et vérification d'un message*

Message reçu	Somme des bits	Message décodé
11101101	6	1110110
10110110	5	<i>Erreur de transmission</i>

**Questions :**

- Combien / quels types d'erreurs peuvent être détectés ?
- Est-ce que l'une des variantes (paire / impaire) est supérieure à l'autre ?

# Caractéristique d'un code

Un **codage par blocs** découpe un message en blocs de taille fixe.  
Il est caractérisé par :

- le nombre de caractères  $k$  du message effectif
- le nombre de caractères  $r$  de redondance
- le nombre de caractères  $n = k + r$  du message codé

On parle de  $(n, k)$ -codes

**Rendement** :  $R = \frac{k}{n}$

Un code est  **$t$ -détecteur / correcteur** s'il permet de détecter / corriger toute erreur affectant  $t$  caractères ou moins.

*Exemple* : Bit de parité :

- $k = 7$ ,  $r = 1$ , donc  $n = 8$ , rendement :  $R = \frac{7}{8}$
- Le code est 1-détecteur et 0-correcteur  
(détecte aussi 3 erreurs, mais pas 2  $\rightsquigarrow$  pas 3-détecteur)

# Détection / correction d'erreurs

*En général* : les messages sont des mots  $m \in A^*$  sur un alphabet  $A$   
Nous distinguons un ensemble  $C \subseteq A^*$  de *codes valides*

*Exemples* :

- RIB : codes “divisibles par 97”
- Bit de parité : Octet avec parité (im)paire

**Détection d'erreurs** : Capacité de dire, pour un  $m'$  reçu, si  $m' \in C$

**Correction d'erreurs** : Capacité de trouver, pour un  $m'$  reçu, un  $m \in C$  approprié

# Plan

4

## Codes correcteurs et détecteurs d'erreurs

- Motivation
- Détection d'erreurs : Notions et exemples de base
- **Détection d'erreurs : CRC**

# Contrôle de Redondance Cyclique (CRC)

## Principes :

- A plusieurs égards : une généralisation du bit de parité.
- Principalement pour la *détection* d'erreurs ; certaines variantes aussi pour la *correction*

**Deux vues :** Représentation des opérations courantes (codage / décodage ...) comme

- opérations sur des polynômes à coefficients dans  $\mathbf{Z}/2\mathbf{Z}$ .
- manipulation de séquences de bits

## De nombreuses applications :

- Protocoles de communication : USB, Bluetooth, Ethernet, CAN, FlexRay
- Compression : Gzip, Bzip2, PNG
- Systèmes de stockage de masse et de fichiers : ext4, Btrfs

# Rappel : Polynômes

Voir aussi le Chapitre 2 du module Mathématiques du L1 :

<http://moodle.univ-tlse3.fr/course/view.php?id=1278>

Un **polynôme** à coefficients dans un corps  $\mathbb{K}$  (noté  $\mathbb{K}[X]$ ) est une expression de la forme

$$P(X) = a_n X^n + a_{n-1} X^{n-1} + \dots + a_2 X^2 + a_1 X + a_0$$

avec  $n \in \mathbf{N}$  et  $a_n \dots a_0 \in \mathbb{K}$

Vous connaissez surtout les corps  $\mathbb{R}$  (nombres réels) et  $\mathbb{C}$  (nombres complexes).

Pour deux polynômes  $A, B \in \mathbb{K}[X]$ , vous maîtrisez :

- l'addition, soustraction et multiplication de  $A$  et  $B$
- la division avec reste avec l'algorithme d'Euclide, qui fournit deux polynomes  $Q, R \in \mathbb{K}[X]$  tq.  $A = QB + R$  et  $\deg(R) < \deg(B)$

# Rappel : $\mathbf{Z}/2\mathbf{Z}$

Nous utiliserons un corps  $\mathbb{K}$  spécifique,  $(\mathbf{Z}/2\mathbf{Z}, +, *)$ , avec

- le support  $\mathbf{Z}/2\mathbf{Z} = \{0, 1\}$
- des opérations d'addition  $+$  et multiplication  $*$
- l'inverse de l'addition " $-$ " tq. pour tout  $x : x + (-x) = 0$
- l'inverse de la multiplication " $^{-1}$ " tq. pour tout  $x \neq 0 : x * x^{-1} = 1$

+	0	1
0	0	1
1	1	0

*	0	1
0	0	0
1	0	1

## Questions

- Calculez  $-0, -1, 1^{-1}$
- Si vous interprétez 0 et 1 comme des valeurs de vérité "faux" et "vrai", à quelle opération sur les Booléens correspondent  $+$  et  $*$  ?

# Polynômes sur $\mathbf{Z}/2\mathbf{Z}$ et séquences de bits

Les **polynômes sur  $\mathbf{Z}/2\mathbf{Z}$**  ont uniquement des coefficients 0 ou 1.

*Exemple :*

$$P(X) = 1X^7 + 1X^6 + 0X^5 + 0X^4 + 0X^3 + 1X^2 + 0X^1 + X^0 = X^7 + X^6 + X^2 + X^0$$

**Représentation comme séquence de bits :**

- Un polynôme sur  $\mathbf{Z}/2\mathbf{Z}$  peut être représenté comme mot de ses coefficients  $a_n a_{n-1} \dots a_0$ , avec  $a_i \in \{0, 1\}$

*Convention :* ordre décroissant des indices des coefficients

- Pour avoir une représentation, il faut fixer la longueur du mot

*Exemple :*  $P(X)$  peut être représenté comme

- 11000101 (pour  $n = 7$ )
- 00011000101 (pour  $n = 10$ )
- La représentation comme séquence de bits permet de déterminer le polynôme. *Exemple :* 01001100 correspond à  $x^6 + x^3 + x^2$

# Opérations sur les polynômes sur $\mathbf{Z}/2\mathbf{Z}$ (1)

## Addition :

- Polynômes : position par position
- Mots de bits : position par position sur des mots de même longueur (si nécessaire, remplir avec des 0 à gauche).

*Attention* : pas de retenue, ce n'est pas une addition binaire !

- Addition des nombres 5 et 3 en binaire :  $(101)_2 + (11)_2 = (1000)_2$
- Addition des polynômes  $X^2 + 1$  et  $X + 1$  en  $\mathbf{Z}/2\mathbf{Z}$  :  
 $[101] + [11] = [110]$  correspond à  $X^2 + X$

## Soustraction :

Montrez : pour deux polynomes  $P(X)$  et  $Q(X)$  (sur  $\mathbf{Z}/2\mathbf{Z}$ !!)

$$P(X) - Q(X) = P(X) + Q(X)$$

**Calculez**  $A(X) + B(X)$  et  $A(X) - B(X)$

pour :  $A(X) = x^3 + x^2 + 1$ ,  $B(X) = x^4 + x^2 + x^1$

Opérations sur les polynômes sur  $\mathbb{Z}/2\mathbb{Z}$  (2)

**Multiplication** d'un polynôme  $P(X) = a_n X^n + \dots + a_1 X + a_0$  et d'un monome  $X^k$  (avec  $k \geq 0$ ) :

- Polynômes :  $P(X) * X^k = a_n X^{n+k} + \dots + a_1 X^{1+k} + a_0 X^k$
- Mots de bits : décaler de  $k$  bits à gauche, remplir avec des 0 à droite

**Calculez** :  $A(X) * X^2$  et  $B(X) * X^0$

**Multiplication** de deux polynômes  $P(X) = a_n X^n + \dots + a_1 X + a_0$  et  $Q(X) = b_m X^m + \dots + b_1 X + b_0$  :

$$P(X) * Q(X) = P(X) * b_m X^m + \dots P(X) * b_1 X + P(X) * b_0$$

**Calculez**  $A(X) * B(X)$

Opérations sur les polynômes sur  $\mathbb{Z}/2\mathbb{Z}$  (3)

**Division** : Ici : algorithme pour n'importe quel  $A, B \in \mathbb{K}[X]$

*Entrée* : Polynomes  $A, B \in \mathbb{K}[X]$

*Sortie* : Polynomes  $Q, R \in \mathbb{K}[X]$  tq.  $A = QB + R$  et  $\deg(R) < \deg(B)$

*Notation* :  $Q = A \div B$  et  $R = A \bmod B$

**begin**

$Q := 0;$                      $R := A;$

$d := \deg(B);$      $c := \text{coeff}(B);$

**while**  $\deg(R) \geq d$  **do**

$S := (\text{coeff}(R)/c) * (X ** (\deg(R) - d))$

$Q := Q + S;$

$R := R - S * B;$

**end while**

**return**  $(Q, R)$

**end**

*Fonctions auxiliaires* :  $\deg$  degré ;  $\text{coeff}$  coefficient dominant

Opérations sur les polynômes sur  $\mathbb{Z}/2\mathbb{Z}$  (4)

Exemple : Division de  $A = X^4 + X^2 + X^1$  par  $B = X + 1$

		S	Q
R	$X^4 + X^2 + X^1$		
- S * B	$-(X^4 + X^3)$	$X^3$	$X^3$
R	$X^3 + X^2 + X^1$		
- S * B	$-(X^3 + X^2)$	$X^2$	$X^3 + X^2$
R	$X^1$		
- S * B	$-(X^1 + 1)$	1	$X^3 + X^2 + 1$
R	1		

Résultat :  $Q = (X^3 + X^2 + 1)$  et  $R = 1$

Vérifiez que  $A = (X^3 + X^2 + 1) * B + 1$

Opérations sur les polynômes sur  $\mathbb{Z}/2\mathbb{Z}$  (4)

*Exemple* : Division de  $A = X^4 + X^2 + X^1$  par  $B = X + 1$   
avec des séquences de bit : division de [10110] par [11]

Algorithme programmé en TP

		S	Q
R	1 0 1 1 0		
- S * B	- 1 1	$1 * X^3$	1
R	1 1 1 0		
- S * B	- 1 1	$1 * X^2$	11
R	0 1 0		
- S * B	1 1	$0 * X^1$	110
R	1 0		
- S * B	- 1 1	$1 * X^0$	1101
R	1		

*Résultat* : Q = [1101], R = [1]

**Exercice** : calculez la division de [1011000] par [101]

# CRC : Principe (1)

*Désormais :*

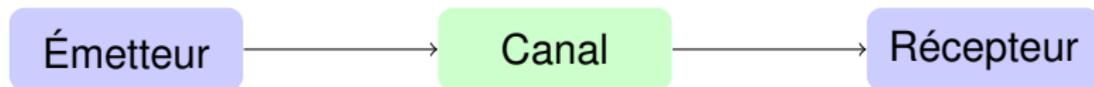
- nous considérons uniquement des polynômes sur  $\mathbf{Z}/2\mathbf{Z}$
- nous faisons l'amalgame entre un polynôme et sa représentation (séquence de bits)

**Paramètre :**

- $G(X)$  : polynôme générateur de degré  $n$ , partagé entre émetteur et récepteur.

Différents  $G(X)$  ont des capacités de détection d'erreur différentes

**Question :** Est-ce que le message codé  $Env(X)$  envoyé par l'émetteur est le même que le message  $Rec(X)$  reçu par le récepteur ?



## CRC : Principe (2)

**Codage** pour envoyer un message  $M(X)$  :

- Calculer :  $R(X) = (M(X) * X^n) \bmod G(X)$
- Le message envoyé :  $Env(X) = M(X) * X^n + R(X)$

*Observations :*

- il existe  $Q(X)$  tel que  $M(X) * X^n = Q(X) * G(X) + R(X)$
- $Env(X) = M(X) * X^n + R(X) = Q(X) * G(X)$
- donc :  $Env(X) \div G(X) = M(X) * X^n$  et  $Env(X) \bmod G(X) = 0$

**Décodage** pour détecter une erreur de transmission

- Soit  $Rec(X)$  le message reçu.
- Si  $Rec(X) \bmod G(X) = 0$ , probablement  $Rec(X) = Env(X)$   
Récupérer  $M(X) = Rec(X) \div X^n$
- Si  $Rec(X) \bmod G(X) \neq 0$ , il y a certainement une erreur

# CRC : Exemple de codage

- *Générateur* :  $G(X) = X^2 + 1$  [101]  
polynôme de degré  $n = 2$
- *Message à coder* :  $M(X) = X^4 + X^2 + X$  [10110]
- Calculer  $R(X) = (M(X) * X^2) \bmod G(X) = X$   
[1011000] mod [101] = [10]
- Message envoyé :  
 $Env(X) = M(X) * X^2 + R(X) = X^6 + X^4 + X^3 + X$   
[1011000] + [10] = [1011010]

# CRC : Exemple de décodage

*Rappel :*

$$G(X) = X^2 + 1 \text{ [101]} \text{ et } Env(X) = X^6 + X^4 + X^3 + X \text{ [1011010]}$$

Soit  $Rec_1(X) = X^6 + X^4 + X^3 + X$  [1011010] le message reçu.

- $Rec_1(X) = (X^4 + X) * G(X)$ , donc  $Rec_1(X) \bmod G(X) = 0$   
 $\rightsquigarrow$  pas d'erreur de transmission

Soit  $Rec_2(X) = X^6 + X^5 + X^3 + X$  [1101010] le message reçu.

- $Rec_2(X) \bmod G(X) = X + 1$   
 $\rightsquigarrow$  erreur de transmission

Soit  $Rec_3(X) = X^6 + X^4 + X^2 + 1$  [1010101] le message reçu.

- $Rec_3(X) = (X^4 + 1) * G(X)$ , donc  $Rec_3(X) \bmod G(X) = 0$   
 $\rightsquigarrow$  erreur de transmission non détectée

# CRC : Limites

**Polynôme d'erreur**  $Err(X) = Rec(X) - Env(X)$

*Observation* :  $Err(X) \bmod G(X) = Rec(X) \bmod G(X)$  **Vérifiez !**

- Si  $Rec(X) \bmod G(X) \neq 0$ , CRC diagnostique une erreur.  
Puis,  $Err(X) \bmod G(X) \neq 0$ , donc  $Err(X) \neq 0$   
 $\rightsquigarrow$  toute erreur diagnostiquée l'est effectivement, pas de fausses alarmes

*Ex.* :  $Err_2(X) = Rec_2(X) - Env(X) =$   
 $[1101010] - [1011010] = [0110000]$

- Si  $Rec(X) \bmod G(X) = Err(X) \bmod G(X) = 0$ , alors
  - $Err(X) = 0 \rightsquigarrow$  absence d'erreur
  - ou  $Err(X) \neq 0$  est multiple de  $G(X) \rightsquigarrow$  erreur non détectée  
*Ex.* :  $Err_3(X) = Rec_3(X) - Env(X) =$   
 $[1010101] - [1011010] = [0001111] = G(X) * (X + 1)$