Université de Lille

FACULTÉ DES SCIENCES ET TECHNOLOGIES
Département Informatique

Licence mention informatique

Codage de l'information

année universitaire 2018-2019

Introduction

Objectifs du cours

Le cours de Codage de l'Information est un cours proposé aux étudiants en deuxième année de licence d'informatique.

Il a pour but d'initier les étudiants à différentes problématiques du codage de l'information : comment représenter en machine un nombre réel quelconque ? comment transmettre une information à la sonde Rosetta en s'assurant qu'elle sera décodée correctement ? comment les compresseurs de fichiers fonctionnent-ils ?

Dans la vie réelle une même information peut être représentée de diverses manières : par exemple une « table » pourra notamment être désignée par « mesa », « tisch », « tafel », « pöytä », sans même évoquer les représentations possibles en dessin, en langue des signes, en braille, ...

Afin d'assurer une interopérabilité entre différents systèmes informatiques, il est nécessaire de se mettre d'accord sur une même manière de représenter des nombres (entiers ou réels) et des caractères. Ce sera l'objet de la première partie du cours.

Ces représentations nous amèneront à généraliser cette approche consistant à représenter un symbole ou une valeur par une suite de bits, par exemple. Nous verrons donc ce qu'est un **code** ainsi que certaines familles particulières de codes.

De nombreuses applications nécessitant d'utiliser le moins de mémoire possible, nous verrons comment utiliser les codes ayant une **longueur moyenne** la plus faible possible. Ce qui nous conduire à aborder la **compression de données**.

Enfin, les communications à grande distance (comme sur Internet, par exemple) nécessitent des systèmes pour corriger les erreurs de transmission qui peuvent subvenir. Nous verrons donc comment ajouter le moins d'information possible de manière à ce qu'on puisse détecter voire corriger une erreur de transmission.

Pour résumer, les différents aspects abordés durant ce cours seront :

1. la représentation des données ;
2. les codes ;
3. l'optimalité des codages ;
4. la détection et la correction d'erreurs.

Le poly du cours

Ce polycopié est une compilation de notes de cours de Codage. Il est encore incomplet, et doit très certainement contenir des erreurs. Merci de signaler ces erreurs à l'adresse `Mikael.Salson@univ-lille.fr`.

Ce polycopié n'est pas auto-suffisant, et ne permet certainement pas de se dispenser des séances de cours et de TD. Il devrait évoluer l'année prochaine par la correction des erreurs qu'il peut contenir, et par la complétion de certaines sections.

Ce polycopié est issu d'une version réalisée par Éric Wegrzynowski, elle-même complètement refondue d'une version précédente qui avait été initiée par Caroline Fontaine qui a participé à ce cours durant les années 1999-2005.

Le site du cours

Le site du cours se trouve à l'adresse :

`http://www.fil.univ-lille1.fr/portail/ls3/codage`

.

Vous pouvez y accéder à l'aide de votre « téléphone élégant » (smartphone pour les anglophones) grâce au QR-code de la figure 1.

FIGURE 1 – QR-code du site du cours

Équipe pédagogique 2018-2019

Francesco de Comité, Dimitri Gallois, Laurent Noé, Mikaël Salson, Éric Wegrzynowski, Léopold Weinberg.

Licence Creative Commons

Chapitre 1

Représentation des nombres

Il y a `10` *sortes de personnes : celles qui comprennent le binaire, et les autres.*
If only `dead` *persons can understand hexadecimal, how many persons understand hexadecimal ?*

1.1 Numération de position

1.1.1 Numération de position

Exemple 1 :
Calculons l'écriture binaire de l'entier $n = 2395$. Pour cela procédons par divisions euclidiennes successives jusqu'à obtenir un quotient plus petit que $b = 2$ (autrement dit 1).

$$\begin{aligned}
2395 &= 2 \times 1197 + 1 \\
1197 &= 2 \times 598 + 1 \\
598 &= 2 \times 299 + 0 \\
299 &= 2 \times 149 + 1 \\
149 &= 2 \times 74 + 1 \\
74 &= 2 \times 37 + 0 \\
37 &= 2 \times 18 + 1 \\
18 &= 2 \times 9 + 0 \\
9 &= 2 \times 4 + 1 \\
4 &= 2 \times 2 + 0 \\
2 &= 2 \times 1 + 0
\end{aligned}$$

L'écriture binaire du nombre $n = 2395$ est donc

$$\overline{\texttt{100101011011}}_2.$$

Exemple 2 :
Calculons maintenant l'écriture du même nombre en base $b = 8$ par divisions euclidiennes

successives jusqu'à obtenir un quotient inférieur à 8.

$$2395 = 8 \times 299 + 3$$
$$299 = 8 \times 37 + 3$$
$$37 = 8 \times 4 + 5$$

L'écriture octale du nombre $n = 2395$ est donc

$$\overline{\texttt{4533}}_8.$$

Exemple 3 :
Calculons enfin l'écriture du même nombre en base $b = 16$ par divisions euclidiennes successives jusqu'à obtenir un quotient inférieur à 16.

$$2395 = 16 \times 149 + 11$$
$$149 = 16 \times 9 + 5$$

L'écriture hexadécimale du nombre $n = 2395$ est donc

$$\overline{\texttt{95B}}_{16}.$$

(Notons l'emploi du chiffre `B` pour représenter l'entier 11.)

1.1.2 Algorithme de conversion

Écriture en base b d'un entier n : Le calcul des chiffres de l'écriture d'un entier naturel n en base b s'effectue par des divisions euclidiennes successives par b, en partant de l'entier n jusqu'à obtenir un quotient plus petit que b. L'algorithme 1.1 montre comment obtenir cette écriture. Les lignes 3 et 4 correspondent au calcul du reste et du quotient dans la division euclidienne de l'entier m par b. Dans certains langages de programmation, on peut traduire cela en une seule instruction effectuant une seule division.

Algorithme 1.1 Calcul de l'écriture en base b d'un entier naturel n

Entrée : b la base de numération, n un entier naturel.
Sortie : $x_0, x_1, \ldots x_{p-1}$ les chiffres de l'écriture de n en base b.

1: $m := n$, $i := 0$
2: **tant que** $m \geq b$ **faire**
3: $\quad r := m \pmod b$
4: $\quad m := m \div b$
5: $\quad x_i :=$ chiffre correspondant à r
6: $\quad i := i + 1$
7: **fin tant que**
8: $x_i :=$ chiffre correspondant à m
9: **renvoyer** $x_0, x_1, \ldots, x_i$.

Les exemples 1, 2 et 3 sont des illustration parfaites de l'exécution de cet algorithme avec $n = 2395$ dans les trois cas et $b = 2$, $b = 8$ et $b = 16$.

Calcul d'un entier à partir de son écriture dans une base : Inversement à partir de la connaissance de l'écriture $\overline{\mathtt{x_{p-1}x_{p-2}\dots x_1x_0}}_2$ dans une base b d'un entier n, il est possible de calculer cet entier par une évaluation de la somme

$$n = \sum_{k=0}^{p-1} x_k b^k.$$

L'algorithme 1.2 est un algorithme évaluant une telle somme.

Algorithme 1.2 Calcul d'un entier n à partir de son écriture dans une base b

Entrée : b une base, x_0, x_1, …, x_{p-1} les chiffres de l'écriture en base b d'un entier.
Sortie : l'entier n dont on a eu l'écriture en base b.
1: $n :=$ l'entier correspondant au chiffre x_0
2: **pour** k variant de 1 à $p-1$ **faire**
3: $\quad n := b \times n +$ l'entier correspondant au chiffre x_k
4: **fin pour**
5: **renvoyer** n

Remarque : L'algorithme 1.2 est un cas particulier d'un algorithme plus général d'évaluation d'un polynôme en un point : l'algorithme de Horner.

1.1.3 Taille d'un entier

Intuitivement, on comprend que plus un nombre entier est grand, plus longue sera son écriture, et ceci quelque soit la base utilisée pour cette écriture. De même, on conçoit bien que l'écriture d'un nombre est d'autant plus grande que la base est petite. Nous allons donc tenter de répondre ici aux deux questions suivantes :

1. comment varie le nombre de chiffres de l'écriture d'un nombre entier lorsque cet entier croît ?
2. quel est le rapport entre le nombre de chiffres de l'écriture d'un nombre entier dans une base, et celui de l'écriture du même entier dans une autre base ?

Commençons par définir la taille d'un entier. Nous appellerons *taille de l'écriture en base b d'un entier $n \in \mathbb{N}$*, ou plus simplement *taille de l'entier n en base b*, le nombre minimal de chiffres nécessaires à l'écriture de cet entier en base b, c'est–à–dire le plus petit entier p tel qu'il existe p entiers x_0, x_1, …, x_{p-1} compris entre 0 et $b-1$ tels que

$$n = \sum_{k=0}^{p-1} x_k b^k.$$

Pour un entier n non nul, la minimalité de p implique que $x_{p-1} \neq 0$. Autrement dit la taille de n est la longueur de l'écriture de n en base b sans zéros superflus à gauche.

Nous noterons $|n|_b$ cette taille.

Partons d'un premier constat.

Lemme 1.1. *Soit $p \geq 1$ un entier.*

Un entier n a une taille égale à p si et seulement s'il appartient à l'intervalle $[\![b^{p-1}, b^p - 1]\!]$. Autrement dit,

$$|n|_b = p \iff n \in [\![b^{p-1}, b^p - 1]\!].$$

À partir de ce lemme nous pouvons en déduire que

$$\begin{aligned} |n|_b = p &\iff b^{p-1} \leq n < b^p \\ &\iff (p-1)\ln b \leq \ln n < p \ln b \\ &\iff p - 1 \leq \frac{\ln n}{\ln b} < p \end{aligned}$$

La seconde ligne est justifiée parce que la fonction ln est strictement croissante, et la troisième parce que $\ln b > 0$ puisque $b \geq 2$.

Puisque $p-1$ et p sont deux entiers consécutifs, on déduit de la troisième ligne que la partie entière de $\log_b n = \frac{\ln n}{\ln b}$ est égale à $p-1$ et par conséquent, en notant $\lfloor x \rfloor$ la partie entière d'un réel x, on a

$$p = \lfloor \log_b n \rfloor + 1.$$

Théorème 1.1. *Pour tout entier $n > 0$ et toute base $b \geq 2$, on a*

$$|n|_b = \lfloor \log_b n \rfloor + 1.$$

Ce théorème permet de quantifier les intuitions que nous pouvons avoir :

— Plus un entier grandit, plus la taille de son écriture augmente. La fonction $\log_b$ est en effet strictement croissante et tend vers l'infini quand la variable tend vers l'infini. Mais on sait que la croissance de cette fonction est bien moins rapide que celle de la variable ($\lim_{x\to+\infty} \frac{\log_b x}{x} = 0$). On peut aussi simplifier l'expression de la taille d'un grand entier par en se contentant de l'approximer par $\log_b n$, puisque pour $n \to +\infty$

$$|n|_b \sim \log_b n.$$

— Pour un même entier ses écritures sont d'autant plus courtes que la base est grande. En effet si $b > b'$, pour tout réel x, $\log_b x < \log_{b'} x$ et donc $|n|_b \leq |n|_{b'}$.

— On peut même quantifier le rapport des tailles dans deux bases différentes d'un grand entier :

$$\begin{aligned} \frac{|n|_b}{|n|_{b'}} &= \frac{\lfloor \log_b n \rfloor + 1}{\lfloor \log_{b'} n \rfloor + 1} \\ &\sim \frac{\log_b n}{\log_{b'} n} \\ &= \frac{\ln b'}{\ln b}. \end{aligned}$$

Ce dernier point montre que le rapport des tailles d'un grand entier dans deux bases différentes est approximativement égal au rapport des logarithmes de ces deux bases.
Par exemple, avec $b = 2$ et $b' = 10$, on a

$$\frac{|n|_2}{|n|_{10}} \approx \frac{\ln 10}{\ln 2} = \log_2 10 \approx 3,332.$$

1.2 Représentation des nombres en informatique

1.2.1 Bases courantes en informatique

En informatique les bases les plus utilisées sont le binaire, l'octal, et l'hexadécimal. Le binaire est utilisé pour des raisons évidentes, un bit (*binary digit*) étant la quantité minimale d'information pouvant être transmise.

L'octal sert dans le cadre d'une représentation par paquets de 3 bits. Dans les années 1960, des ordinateurs travaillaient avec des registres de 12 ou 18 bits (par exemple le DEC PDP-8)

De nos jours, l'hexadécimal est particulièrement utilisé car il sert à représenter des paquets de 4 bits : les processeurs actuels travaillent avec des registres de 8, 16, 32 ou 64 bits.

La table 1.1 montre la représentation des 16 premiers entiers naturels dans ces trois bases ainsi que dans la base décimale, couramment utilisée dans la vie quotidienne.

Déc.	Hexa.	Octal	Bin.
00	0	00	0000
01	1	01	0001
02	2	02	0010
03	3	03	0011
04	4	04	0100
05	5	05	0101
06	6	06	0110
07	7	07	0111
08	8	10	1000
09	9	11	1001
10	A	12	1010
11	B	13	1011
12	C	14	1100
13	D	15	1101
14	E	16	1110
15	F	17	1111

TABLE 1.1 – Écriture des entiers de 0 à 15 en décimal, hexadécimal, octal et binaire

1.2.2 Autres représentations

Bien que la représentation de nombres en binaire, octal ou hexadécimal soit courante, il ne s'agit pas de la seule représentation possible. D'autres représentations ont été utilisées ou continuent à être utilisées.

Biquinaire

Le biquinaire permet la représentation de chiffres décimaux. Cette représentation, avant d'être employée dans des ordinateurs, l'a été dans des bouliers depuis environ deux-mille ans.

La représentation biquinaire, comme son nom l'indique, se décompose en deux éléments. Un élément à deux états (*binaire*) et un élément à cinq états (*quinaire*). L'élément à cinq états est utilisé pour coder les chiffres de 0 à 4. L'élément à deux états sert à pouvoir ajouter 5 au chiffre représenté par le composant quinaire. Il est donc possible de représenter tous les chiffres compris entre 0 et 9.

La représentation biquinaire peut se faire sur 7 bits, dans ce cas l'élément à deux états est sur deux bits (un seul est à 1), et l'élément à cinq états est sur cinq bits (dont un seul est à 1). Voir des exemples dans la TABLE 1.2 page suivante.

Binaire codé décimal (BCD)

La représentation BCD utilise un nombre fixe de bits pour représenter chacun des chiffres décimaux par leur représentation binaire. Le nombre minimal de bits afin de représenter dix valeurs étant de quatre [1], un chiffre en BCD est représenté sur quatre bits, quelque soit sa valeur (voir TABLE 1.2).

L'addition de deux nombres BCD peut donner des résultats semblant absurdes. Par exemple si on additionne 7 et 8 en représentation BCD, cela nous donne $0111 + 1000 = 1111$. Or 1111 n'est pas un chiffre en BCD. Dans une telle situation, lorsque le résultat dépasse le plus grand chiffre représentable (1001), il faut ajouter six (0110) au chiffre dépassant la capacité. Dans le cas présent cela nous donne $1111 + 0110 = 10101$ qui, remis sur un multiple de quatre bits, correspond à 00010101, soit la représentation de 15 en BCD.

Déc.	Biquinaire	BCD
0	0100001	0000
1	0100010	0001
2	0100100	0010
3	0101000	0011
4	0110000	0100
5	1000001	0101
6	1000010	0110
7	1000100	0111
8	1001000	1000
9	1010000	1001

TABLE 1.2 – Représentations biquinaire et BCD des chiffres décimaux.

Le BCD est notamment utilisé dans certains afficheurs à cristaux liquides pour représenter les nombres décimaux affichés.

1.2.3 Entiers de précision illimitée

Certains langages de programmation, tel Python, fournissent des entiers de précision illimitée permettant de représenter des entiers arbitrairement grands. La seule limite à la taille de ces nombres est due à la quantité de mémoire disponible sur la machine. Les opérations sur des entiers de précision illimitée sont moins efficaces que sur des entiers tenant dans un registre mémoire. En effet dans le second cas les opérations arithmétiques sont directement encodées dans le processeur alors que pour des entiers de précision illimitée il est nécessaire d'effectuer le calcul côté logiciel.

1.3 Entiers signés

Nous venons de voir différentes représentations pour des entiers non signés, c'est-à-dire pour des entiers positifs. Nous allons présenter deux techniques permettant également la représentation d'entiers négatifs.

1. Puisque $\lfloor \log_2(10) \rfloor + 1 = 4$.

1.3.1 Représentation en signe + valeur absolue

Pour un entier représenté sur n bits, les $n-1$ bits de poids faible sont réservés à la représentation de la valeur absolue de l'entier tandis que le bit de poids fort permet d'indiquer le signe de cet entier. Le bit de poids fort est à 1 si et seulement si l'entier est négatif.

1.3.2 Représentation en complément à deux

La représentation en complément à deux sur n bits se fait de la façon suivante :
— si l'entier est positif, sa valeur binaire est stockée ;
— sinon, c'est la valeur de $2^n + i$ qui est stockée.

On peut formuler cela d'une autre manière. Lorsque l'entier i à stocker est négatif, il faut prendre la représentation binaire de $-i$ sur n bits, calculer le complémentaire (ce qui consiste à inverser la valeur de chacun des bits) et d'ajouter 1 au résultat.

Sur n bits, cette représentation permet de stocker tous les entiers compris entre -2^{n-1} et $2^{n-1} - 1$. Dans la représentation en complément à 2, le bit de poids fort indique le signe de l'entier, comme dans la représentation signe-valeur absolue.

Exemple 4 :

Représentation de l'entier -73 sur 8 bits :

Première méthode	Seconde méthode
$2^8 - 73 = 183 = \overline{\texttt{10110111}}_2$	$73 = \overline{\texttt{01001001}}_2$
	Complément : $\overline{\texttt{10110110}}_2$
	Ajout de 1 : $\overline{\texttt{10110111}}_2$

1.4 Nombres flottants

Les nombres à virgule flottante, désignés plus souvent sous l'appellation plus concise *nombres flottants* (*floating-point numbers*), sont en général des nombres non entiers dont on écrit les chiffres uniquement après la virgule et auquel on ajoute un exposant pour préciser de combien de positions la virgule doit être déplacées.

Par exemple en notation décimale, on écrira le nombre $31,41592$ sous la forme

$$3,141592 \times 10^1,$$

et le nombre $-0,01732$ sous la forme

$$-1,732 \times 10^{-2}.$$

Dans les deux cas, on appelle *exposant* la puissance, et *mantisse* le nombre qui multiplie le nombre 10 à la puissance de l'exposant ($3,141592$ dans le premier cas, et $1,732$ dans le second).

Cette écriture permet dans un espace limité pour le nombre de chiffres d'écrire des nombres d'ordres de grandeur très variables, ce que ne permettrait pas une écriture dans laquelle la virgule serait toujours placée entre le chiffre des unités ou dernier chiffre de la partie entière et le premier chiffre de la partie fractionnaire.

En informatique, une norme s'est imposée pour la représentation des nombres flottants. C'est la norme IEEE 754.

1.4.1 La norme IEEE 754

Dans la norme IEEE 754, un nombre flottant est toujours représenté par un triplet

$$(s, e, m)$$

où

— la première composante s détermine le signe du nombre représenté, ce signe valant 0 pour un nombre positif, et 1 pour un nombre négatif ;
— la deuxième e désigne l'exposant ;
— et la troisième m désigne la mantisse.

La norme établit deux bases possibles pour la représentation : $b = 2$ ou $b = 10$. La norme prévoit sept formats possibles :

— quatre formats pour la représentation en base $b = 2$: sur 16, 32, 64 ou 128 bits [2] ;
— et trois formats pour la base $b = 10$: sur 32, 64 et 128 bits.

Dans la suite nous ne mentionnons que les formats de base $b = 2$.

Le nombre représenté par le triplet (s, e, m) est

$$(-1)^s \times 2^e \times m.$$

Ainsi, les deux nombres précédents sont représentés par les triplets $(0, 4, 1.963495)$ (car $31,41592 = 1,963495 \times 2^4$) et $(1, -6, 1.10848)$ (car $-0,01732 = -1,10848 \times 2^{-6}$).

Il reste à coder les trois composantes (s, e, m) dans un format binaire.

Le signe : La représentation du signe est simple, il suffit d'un bit pour le représenter : le bit 0 pour le signe + et le bit 1 pour le signe −. Ce bit de signe est le bit le plus significatif de la représentation, c'est–à–dire le bit le plus à gauche.

L'exposant : Le nombre de bits réservés pour la représentation de l'exposant et celle de la mantisse dépend des formats. Le tableau 1.3 donne pour chacun des formats, le nombre w de bits réservés à la représentation de l'exposant, et le nombre t de bits réservés pour celle de la mantisse. Dans tous les cas, on a

$$n = 1 + w + t,$$

où n désigne la taille totale de la représentation.

n	16 bits	32 bits	64 bits	128 bits
w	5	8	11	15
t	10	23	52	112

TABLE 1.3 – Paramètres des formats IEEE-754

L'exposant, représenté sur w bits, peut prendre toute valeur entière comprise entre deux entiers e_{min} et e_{max}. Ces deux bornes doivent vérifier la relation $e_{min} = 1 - e_{max}$. De plus, e n'est pas stocké directement, à la place c'est l'entier naturel E qui le codera. On doit avoir

$$1 \leq E \leq 2^w - 2 \text{ et } E = e + 2^{w-1} - 1$$

2. Le type `float` en PYTHON suit la norme IEEE 754 en base $b = 2$ sur 64 bits.

Ces contraintes donnent les expressions suivantes pour les bornes de l'exposant e

$$e_{max} = 2^{w-1} - 1$$
$$e_{min} = 2 - 2^{w-1}.$$

Les valeurs possibles de l'exposant e selon le nombre de bits w de sa représentation, pour un nombre flottant ordinaire, sont donnés par le tableau 1.4.

Format	e_{min}	e_{max}
16 bits ($w = 5$)	-14	$+15$
32 bits ($w = 8$)	-126	$+127$
64 bits ($w = 11$)	-1022	$+1023$
128 bits ($w = 15$)	-16382	$+16383$

TABLE 1.4 – Valeurs possibles de l'exposant selon le format

La représentation de l'exposant sur w bits nuls est réservée à la représentation de ± 0 et de nombres spéciaux qualifiés de « sous-normaux ». La représentation sur w bits à 1 est réservée aux flottants spéciaux $\pm\infty$ et les `NaN`.

La mantisse : La mantisse doit être l'unique nombre réel m vérifiant

$$1 \leq m < 2$$

et

$$x = (-1)^s \times 2^e \times m.$$

Dans la représentation décimale de m, le chiffre à gauche de la virgule est nécessairement un 1. Il est donc inutile de le représenter. Ce n'est donc pas m qui est directement représenté mais un entier naturel M, sur t bits, tel que :

$$m = 1 + 2^{-t} \times M.$$

Autrement dit, le bit de poids fort de M représente la valeur $1/2$ tandis que le bit de poids faible représente la valeur $1/2^t$.

1.4.2 Un exemple

Soit à représenter le nombre réel

$$x = 29,75.$$

Ce nombre peut s'écrire

$$x = (-1)^0 \times 2^4 \times 1,859375.$$

On a donc $s = 0$, $e = 4$ et $m = 1,859375$.

Selon la norme IEEE 754, la représentation de x sur 32 bits s'écrira

1. `0` pour le bit le plus à gauche (le nombre est positif) ;
2. `10000011` pour l'exposant, en effet $E = e + 2^{w-1} - 1 = 4 + 128 - 1 = 131$;
3. et `110111` suivi de 17 fois le bit `0`, puisque $M = 2^t \times (m - 1)$ s'écrit de cette façon sur 23 bits ($m = 1,859375$ s'écrit $m = \overline{1,110111}_2$ car $0,859375 = \frac{1}{2^1} + \frac{1}{2^2} + \frac{1}{2^4} + \frac{1}{2^5} + \frac{1}{2^6}$).

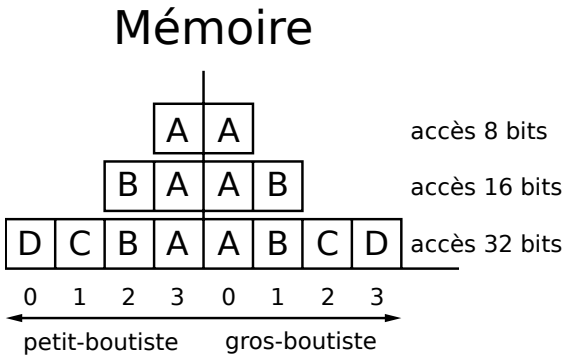


FIGURE 1.1 – Représentation d'une même information en mémoire selon le boutisme.
Source : Konrad Eisele, Wikimedia Commons, CC BY SA

On en déduit que l'écriture binaire de x sur 32 bits est (les espaces sont là pour aider à la lecture du signe, de l'exposant et de la mantisse)

`0` 10000011 11011100000000000000000

ou plus lisiblement en représentation hexadécimale

`41EE0000`.

1.5 Représentation de l'information en mémoire

Les données stockées sur plusieurs octets peuvent être ordonnées en mémoire de plusieurs manières, on appelle cela le *boutisme* (ou *endianness* en anglais).

En gros-boutisme (*big endian*), les octets de poids forts sont stockés aux adresses les plus petites. Autrement dit, pour un entier de quatre octets dont la valeur est $\mathsf{A}\times 2^{24}+\mathsf{B}\times 2^{16}+\mathsf{C}\times 2^{8}+\mathsf{D}$, l'octet A sera stocké à la plus petite adresse puis suivront dans l'ordre des adresses les octets B, C et D (voir FIGURE 1.1). À l'inverse, en petit-boutisme, l'octet D sera stocké à la plus petite adresse puis suivront, dans l'ordre des adresses les octets C, B et A.

Cette gestion de la mémoire est transparente pour l'utilisateur et pour le programmeur, le plus souvent, comme elle est gérée directement par le système. Cependant connaître le boutisme utilisé est important lorsqu'on souhaite regarder les valeurs des octets en mémoire (ou sur disque, c'est équivalent).

À titre d'exemple voici la représentation, dans un éditeur hexadécimal, d'un entier stocké sur 4 octets : `02 04 08 10`. Selon le boutisme, la valeur de l'entier ne sera pas la même. Il est donc important de connaître le boutisme pour bien interpréter la valeur stockée.

1.6 Opérations logiques sur les entiers

Chacun connaît les opérations arithmétiques usuelles sur les nombres entiers : addition, soustraction, multiplication et division.

La représentation binaire des nombres entiers conduit à définir de nouvelles opérations qui s'appuient sur les opérateurs logiques comme la conjonction, la disjonction, le ou-exclusif et la négation.

1.6.1 Les opérations $\vee$, $\wedge$ et $\oplus$

Commençons par rappeler les opérateurs logiques `ou` que nous noterons $\vee$, `et` noté $\wedge$ et `ou-exclusif` noté $\oplus$. Nous rappelons leur définition sous forme de tableaux à deux entrées, en notant `0` la valeur booléenne `Faux`, et `1` la valeur booléenne `Vrai` (cf 1.5).

$\vee$	0	1
0	0	1
1	1	1

$\wedge$	0	1
0	0	0
1	0	1

$\oplus$	0	1
0	0	1
1	1	0

TABLE 1.5 – Tables de vérité des opérateurs booléens $\vee$, $\wedge$ et $\oplus$

Ces trois opérations booléennes peuvent être étendues aux nombres entiers, en les appliquant sur les écritures binaires de ces nombres.

Soient donc n et p deux nombres entiers naturels.

Opération `ou` : l'entier $n \vee p$ est défini comme étant le nombre entier dont la représentation binaire est obtenue en faisant agir l'opérateur $\vee$ bit à bit sur chacun des bits des représentations binaires de n et p. Si ces représentations binaires n'ont pas la même taille, on ajoute des bits nuls superflus en tête de la représentation la plus petite.

Opération `et` : la définition de $n \wedge p$ est analogue.

Opération `ou-exclusif` : idem pour $n \oplus p$.

Exemple 5 :
Calculons $n \vee p$, $n \wedge p$ et $n \oplus p$ avec $n = 21$ et $p = 12$.

En binaire, n s'écrit $n = \overline{\texttt{10101}}_2$, et $p = \overline{\texttt{1100}}_2$. L'écriture binaire de p a un bit de moins que n, on rajoute donc un bit nul en tête. Par conséquent, en posant les opérations

$$\begin{array}{cccccc} & 1 & 0 & 1 & 0 & 1 \\ \vee & 0 & 1 & 1 & 0 & 0 \\ \hline = & 1 & 1 & 1 & 0 & 1 \end{array} \qquad \begin{array}{cccccc} & 1 & 0 & 1 & 0 & 1 \\ \wedge & 0 & 1 & 1 & 0 & 0 \\ \hline = & 0 & 0 & 1 & 0 & 0 \end{array} \qquad \begin{array}{cccccc} & 1 & 0 & 1 & 0 & 1 \\ \oplus & 0 & 1 & 1 & 0 & 0 \\ \hline = & 1 & 1 & 0 & 0 & 1 \end{array}$$

on obtient

$$\begin{aligned} n \vee p &= \overline{\texttt{11101}}_2 = 29 \\ n \wedge p &= \overline{\texttt{00100}}_2 = 4 \\ n \oplus p &= \overline{\texttt{11001}}_2 = 25. \end{aligned}$$

Propriétés de ces opérations

Associativité : ces trois opérations sont associatives. Autrement dit pour tout triplet d'entiers n, p et q, on a

$$\begin{aligned} (n \vee p) \vee q &= n \vee (p \vee q) \\ (n \wedge p) \wedge q &= n \wedge (p \wedge q) \\ (n \oplus p) \oplus q &= n \oplus (p \oplus q). \end{aligned}$$

Commutativité : ces trois opérations sont commutatives. Autrement dit pout tout couple d'entiers n et p, on a

$$n \vee p = p \vee n$$
$$n \wedge p = p \wedge n$$
$$n \oplus p = p \oplus n.$$

Élement neutre : 0 est un élément neutre pour les opérations $\vee$ et $\oplus$. En effet, pour tout entier n on a

$$n \vee 0 = n$$
$$n \oplus 0 = n.$$

Mais l'opération $\wedge$ n'admet pas d'élémént neutre dans $\mathbb{N}$. En revanche, si on se limite à l'intervalle des entiers compris entre 0 et $2^p - 1$ pour un certain $p > 0$, alors l'entier $2^p - 1$ est un élément neutre pour l'opération $\wedge$. En effet pour tout entier $n \in [\![0, 2^p - 1]\!]$, on a

$$n \wedge (2^p - 1) = n.$$

1.6.2 Et la négation ?

Une opération logique n'a pas été évoquée : la négation. Si on note $\neg b$ la négation d'un bit, cette opération se définit très simplement par :

$$\neg 0 = 1$$
$$\neg 1 = 0.$$

Définir cette opération sur n'importe quel entier naturel pose un petit problème. A priori, on pourrait être tenté de définir la négation d'un entier comme étant l'entier obtenu en changeant tous les bits de la représentation binaire de cet entier. Ainsi on aurait par exemple

$$\neg 21 = \neg \overline{\texttt{10101}}_2 = \overline{\texttt{01010}}_2 = 10.$$

Mais on aurait aussi

$$\neg 53 = \neg \overline{\texttt{110101}}_2 = \overline{\texttt{001010}}_2 = 10.$$

L'opération de négation n'a plus la propriété qu'elle possédait sur les bits d'être injective : la négation de deux bits différents donne deux bits différents.

De plus,

$$\neg 10 = \neg \overline{\texttt{1010}}_2 = \overline{\texttt{0101}}_2 = 5,$$

et donc on a

$$\neg\neg 21 = 5 \neq 10.$$

La négation ne possède plus la propriété qu'elle a sur les bits d'être involutive : la négation de la négation d'un bit est ce bit.

Il n'est donc pas possible de définir une opération de négation pour tous les entiers de $\mathbb{N}$ qui possède cette propriété d'être involutive [3].

3. en fait il est possible de le faire en admettant les représentations binaires infinies à gauche : tout entier n est représenté par un mot binaire constitué de son écriture binaire usuelle complété par une infinité de bits nuls à gauche. La négation d'un entier est donc représentée par un mot binaire contenant une infinité de 1 à gauche, qui ne représente pas un entier naturel. On sort du cadre des entiers naturels pour entrer dans celui des entiers diadiques.

En revanche, la négation peut être définie avec toutes les bonnes propriétés attendues si on limite les entiers à un intervalle de la forme $[\![0, 2^p - 1]\!]$, l'entier p pouvant être n'importe quel entier positif. Si $n \in [\![0, 2^p - 1]\!]$, son écriture binaire a une taille au maximum égale à p. Si cette taille est strictement inférieure à p on complète l'écriture binaire avec des 0 à gauche pour obtenir une écriture de taille p. La négation de n est le nombre entier dont l'écriture binaire est celle obtenue en appliquant la négation sur chacun des bits de l'écriture de n.

Exemple 6 :
Avec $n = 21$ et $p = 8$, on a

$$n = \overline{\texttt{00010101}}_2,$$

et

$$\neg n = \overline{\texttt{11101010}}_2 = 234.$$

Mais si on prend $p = 16$, on a alors

$$n = \overline{\texttt{0000000000010101}}_2,$$

et

$$\neg n = \overline{\texttt{1111111111101010}}_2 = 68054.$$

Ainsi la négation d'un entier n'est définie que relativement à une taille de représentation p donnée.

Propriétés : Soit p une taille de représentation fixée.

— Pour tout entier $n \in [\![0, 2^p - 1]\!]$, on a

$$\neg n \in [\![0, 2^p - 1]\!].$$

— Pour tout entier $n \in [\![0, 2^p - 1]\!]$, on a

$$\neg\neg n = n.$$

L'opération de négation est involutive.

— Pour tout entier n et $p \in [\![0, 2^p - 1]\!]$, on a

$$\neg(n \vee p) = (\neg n) \wedge (\neg p)$$
$$\neg(n \wedge p) = (\neg n) \vee (\neg p).$$

1.6.3 Les décalages

Mentionnons encore deux opérations sur les représentations binaires des entiers.

Décalage à droite : le décalage à droite de k positions d'un entier n, noté $n \gg k$, est l'entier dont l'écriture binaire est obtenue en supprimant les k bits de poids faibles de l'écriture binaire de n.

Décalage à gauche : le décalage à gauche de k positions d'un entier n, noté $n \ll k$, est l'entier dont l'écriture binaire est obtenue en ajoutant k bits nuls à droite de l'écriture binaire de n.

Exemple 7 :
Pour $n = 21 = \overline{\texttt{10101}}_2$,

$$n \gg 2 = \overline{\texttt{101}}_2 = 5,$$
$$n \ll 2 = \overline{\texttt{1010100}}_2 = 84.$$

L'opération de décalage à gauche a une interprétation arithmétique simple :

$$n \ll k = n \times 2^k,$$

et celle de décalage à droite

$$n \gg k = n \div 2^k.$$

Ainsi les opérations de décalage correspondent à des multiplications et divisions par des puissances de 2.

1.6.4 Application de ces opérations

Une application courante de ces opérateurs logiques est l'extraction d'un ou plusieurs bits donnés de la représentation binaire d'un entier.

En effet si n et $k \in \mathbb{N}$, l'expression

$$(n \wedge 2^k) \gg k = (n \wedge (1 \ll k)) \gg k$$

vaut le bit de rang k (le $k+1$-ème bit de poids faible) de l'écriture binaire de n.

Dans le détail $1 \ll k = 2^k$ qui s'écrit, en binaire, avec un 1 suivi de k zéros. Cette valeur va servir de **masque**. Les 0 de l'écriture binaire masquent les bits qui ne nous intéressent pas dans la valeur de n. Pour appliquer ce masque il faut utiliser l'opérateur logique $\wedge$ (`et`) car $0 \wedge x$ donne toujours 0. C'est l'opérateur qu'il faut utiliser avec un masque. Ainsi $n \wedge (1 \ll k)$ permet de mettre tous les bits à 0 sauf celui de rang k qui conserve sa valeur. Il reste enfin à décaler ce bit de k rangs vers la droite afin de l'amener au rang 0.

Terminons avec un exemple. Notre valeur n est inconnue mais est composée de 6 bits : $n = \overline{\texttt{b}_5\texttt{b}_4\texttt{b}_3\texttt{b}_2\texttt{b}_1\texttt{b}_0}_2$. Nous souhaitons récupérer le bit $\texttt{b}_4$ de n. Pour cela nous allons faire un masque avec la valeur $\overline{\texttt{10000}}_2 = 2^4 = 1 \ll 4$. Le résultat du `et` entre ces deux nombres est :

$$\overline{\texttt{b}_5\texttt{b}_4\texttt{b}_3\texttt{b}_2\texttt{b}_1\texttt{b}_0}_2 \wedge \overline{\texttt{10000}}_2 = \overline{\texttt{b}_4\texttt{0000}}_2$$

Pour terminer on décale ce résultat de 4 bits vers la droite, ce qui donne :

$$\overline{\texttt{b}_4\texttt{0000}}_2 \gg 4 = \overline{\texttt{b}_4}_2$$

Ne conserver que certains bits (ou mettre à zéro certains bits)

Cette idée peut être généralisée à un nombre quelconque de bits. Un masque sert à isoler les bits qui nous intéressent dans un entier. Toutes les positions à 1 dans le masque correspondent aux positions que l'on souhaite récupérer dans notre entier. Appliquer l'opérateur $\wedge$ entre le masque et l'entier permet d'obtenir le résultat voulu.

Mettre à 1 certains bits

À l'inverse, on peut utiliser un masque pour mettre des 1 à certaines positions. Dans ce cas il faudra appliquer l'opérateur $\vee$ (`ou`) à l'entier et au masque pour forcer ces valeurs.

Ainsi si on applique le masque $\overline{000111}_2$ à $n = \overline{\mathtt{b_5b_4b_3b_2b_1b_0}}_2$ avec l'opérateur $\vee$, on obtient :

$$\overline{\mathtt{b_5b_4b_3b_2b_1b_0}}_2 \vee \overline{100110}_2 = \overline{\mathtt{1b_4b_3}11\mathtt{b}_0}_2$$

Notre masque a imposé des bits à 1 aux positions auxquelles notre masque possédait des 1.

Associer plusieurs masques

On peut associer plusieurs masques afin de cumuler leurs effets. Si on reprend les deux types de masques vus ci-dessus, il est possible de ne conserver que certains bits de notre entier d'origine et de mettre d'autres bits à 1.

Le schéma général sera celui-ci : $(n \wedge mask_0) \vee mask_1$, où $mask_0$ est le masque ne servant à conserver que certains bits de n et $mask_1$ est le masque forçant certains bits à 1. **Attention** : pour éviter d'avoir à retenir la priorité entre les opérateurs $\wedge$ et $\vee$, ou pour éviter de vous tromper, mettez toujours des parenthèses autour de chaque opération.

En reprenant les deux exemples précédents, il est possible de ne conserver que le bit $\mathtt{b}_4$ dans n et de mettre les bits de rang 1, 2 et 5 à 1.

$$(\overline{\mathtt{b_5b_4b_3b_2b_1b_0}}_2 \wedge \overline{10000}_2) \vee \overline{100110}_2 = \overline{\mathtt{1b_4}0110}_2$$

1.7 Exercices

1.7.1 Bases de numération

Exercice 1-1 *Conversions*

1. Convertir le nombre 797 en base 2, 8 et 16.
2. Quel est le nombre entier dont l'écriture en base 16 est $\overline{793}_{16}$?
3. Écrire en base 16 le nombre dont l'écriture en base 4 est $\overline{2231201}_4$ sans le convertir en base 10.
4. Écrire en base 2 le nombre dont l'écriture en base 16 est $\overline{\mathtt{A2B1}}_{16}$ sans le convertir en base 10.

Exercice 1-2 *Question de cours*

Soit $b \geq 2$ un entier.

Question 1 Quels sont les nombres entiers naturels dont l'écriture en base b comprend n chiffres au maximum ? Combien y en a–t–il ?

Question 2 Même question pour les entiers dont l'écriture en base b possède exactement n chiffres.

Exercice 1-3 *Additions et multiplications*

Question 1 Construisez les tables d'addition et de multiplication en base 2, puis effectuez les opérations $\overline{111010}_2 + \overline{1001}_2$ et $\overline{110010}_2 \times \overline{1001}_2$.

pour approfondir **Exercice 1-4**

En base 10, le nombre 1331 est le cube de 11.

Question 1 Vérifiez si cela est également vrai pour la base 16.

Question 2 Montrez que pour toute base $b > 3$, le nombre $\overline{1331}_b$ est un cube.

Exercice 1-5

Soit x le nombre entier naturel dont l'écriture binaire est $\overline{10\ldots01}_2$ (n chiffres 0 encadrés par deux 1, n étant non nul). Comment s'écrivent x^2 et x^3 en base 2 ?

pour approfondir **Exercice 1-6** *Miroirs*

Dans le système de numération de base b, on considère deux entiers naturels non nuls c et d différents de 1 et de b qui vérifient $c + d = b + 1$, avec $b \geq 3$.

Question 1 Quel encadrement des valeurs de c et d peut-on proposer ? Que sont c et d ?

Question 2 Prouvez que les nombres $n = c(b-1)$ et $m = d(b-1)$ s'écrivent avec les mêmes chiffres mais disposés en ordre inverse.

Question 3 Comment s'écrit la somme $n + m$ en base b ?

1.7.2 Représentation des entiers

Exercice 1-7 *Taille d'un entier*

Le plus grand nombre premier [4] connu ce jour (11 juillet 2019) est le 50-ème nombre premier de Mersenne

$$N = 2^{77232917} - 1.$$

Question 1 Comment s'écrit ce nombre en base 2 ?

Question 2 Combien de chiffres comprend l'écriture de ce nombre en base 10 ?

Question 3 Quelle est la taille de ce nombre (en Mo) représenté en binaire ? en BCD ?

pour approfondir **Exercice 1-8**

Python, à l'inverse de beaucoup de langages, permet de représenter des entiers dont la seule limite est fixée par les capacités mémoires de la machine.

Question 1 En supposant qu'un entier est stocké sur le nombre minimal de bits, de combien de bits sera composé un entier occupant 1 Go de mémoire ?

Question 2 Quel est le plus grand entier qu'on peut représenter sur ce nombre de bits ?

Question 3 De combien de chiffres est composé cet entier en décimal ?

Exercice 1-9 *Représentation binaire des entiers signés*

Question 1 Représentez le nombre -103_{10} en binaire sur 8 bits dans le codage signe–valeur absolue et

4. découvert le 03/01/2018, source : `http://primes.utm.edu/largest.html`

en complément à deux.

Question 2 Suivant la représentation, quel est l'entier codé sur 8 bits par 11011001_2 ?

Question 3 Suivant la représentation, quel est l'entier codé sur 9 bits par 011011001_2 ?

Question 4 Lorsque le codage s'effectue sur n bits, quels sont les entiers que l'on peut représenter avec le codage signe–valeur absolue ? avec le codage en complément à deux ?

Exercice 1-10 *Codage BCD*

Codez en BCD les entiers 123 et 78. Puis effectuez en BCD l'addition $123 + 78$.

Exercice 1-11 *Bug de l'an 2010*

Question 1 Le 1er janvier 2010 certains téléphones portables affichaient les SMS reçus avec une date au 1er janvier 2016. Sur ces téléphones, les années étaient affichées sur deux chiffres. Les dates dans les SMS sont codées en BCD. Comment expliquez-vous ce bug ?

pour approfondir **Exercice 1-12** *Les entiers de MAPLE*

Le logiciel de calcul mathématique MAPLE travaille sur des entiers représentés dans une base qui est la plus grande puissance de 10 dont le carré puisse s'écrire dans un registre du processeur.

Question 1 Déterminez cette base pour un processeur 32 bits.

Question 2 Quel intérêt peut avoir une telle représentation ?

1.7.3 Représentation des flottants

Exercice 1-13 *Nombres à virgule flottante selon la norme IEEE 754*

Dans la norme IEEE-754, les nombres flottants peuvent être représentés en base $b = 2$ sous la forme

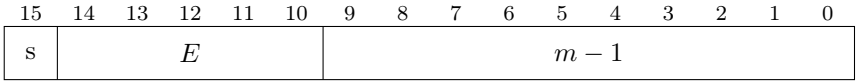
$$x = (-1)^s \times 2^e \times m$$

où m est un réel compris entre 1 inclus et 2 exclu, appelé *mantisse*, et e est un entier relatif, appelé *exposant*.

Il existe quatre codages :

1. sur 16 bits pour une demie précision ;
2. sur 32 bits pour une simple précision ;
3. sur 64 bits pour une double précision ;
4. sur 128 bits pour une quadruple précision.

Dans cet exercice, on va considérer la version sur 16 bits.



— les dix bits de poids faible forment la représentation binaire de la partie fractionnaire de la mantisse, c'est-à-dire de $m - 1$, ainsi le bit de poids faible de la mantisse représente la valeur $\frac{1}{2^{10}}$ alors que le bit de poids fort de la mantisse représente la valeur $\frac{1}{2}$;
— les cinq bits suivants stockent E duquel on obtient l'exposant e en calculant $e = E - 15$;

— et le bit de poids fort indique le signe du nombre (`0` = positif, `1` = négatif).

Question 1 On représente un nombre réel par $\overline{\texttt{B4D2}}_{16}$.

Quelles sont les valeurs de s, E et $m-1$? En déduire la valeur du réel ainsi représenté.

Question 2 Quel est le plus petit réel positif non nul sur 16 bits ? le plus grand ? le deuxième plus grand ?

Exercice 1-14 *Premier entier non représenté par un flottant*

En Python, les entiers sont de précision illimitée, ce qui signifie qu'ils peuvent être aussi grands que la capacité de la machine le permet. En revanche les flottants sont de précision limitée : ils sont stockés sur 64 bits en suivant la norme IEEE-754.

Nous cherchons à trouver le plus petit entier naturel qui ne peut pas être représenté comme un flottant, en suivant la norme IEEE-754. Autrement dit, nous cherchons le plus petit entier `i` tel que l'expression Python suivante soit **vraie** : `i != floor(float(i))`

Question 1 Dans la représentation IEEE-754 sur 64 bits, il existe une seule valeur de l'exposant e pour laquelle tous les flottants positifs ayant cette valeur d'exposant sont uniquement les entiers de 2^e à $2^{e+1}-1$. Quelle est cette valeur de e ?

Question 2 Quel est le plus petit entier qui ne peut pas être représenté par un flottant dans la norme IEEE-754 sur 64 bits ?

1.7.4 Boutisme

Exercice 1-15 *Boutisme*

Question 1 Donnez la représentation (en hexadécimal) de l'entier $2^1+2^{10}+2^{19}+2^{28}$ en gros-boutiste et en petit-boutiste.

1.7.5 Opérations logiques

Exercice 1-16 *Opérations logiques sur les entiers*

Question 1 Calculez $n \oplus m$, $n \wedge m$, $n \vee m$, $n << 2$, $n >> 2$ pour $n = 453$ et $m = 316$, puis donnez les résultats en décimal.

Question 2 L'opération $\oplus$ admet-elle un élément neutre dans $\mathbb{N}$? et $\wedge$? et $\vee$? et si on se restreint à l'intervalle des entiers compris entre 0 et 2^p-1 ?

Question 3 Comment tester si un entier est pair à l'aide des opérations logiques ?

Exercice 1-17 *Le compte est bon*

En utilisant les opérations logiques sur les entiers, arrivez au résultat demandé en utilisant les nombres proposés au plus une fois.

Question 1

Nombres	218	86	15
Résultat		82	

Question 2

Nombres	1	13	23	67
Résultat		111		

Exercice 1-18 *Utilisation de masques*

Soit n un entier représenté sur 8 bits que nous écrirons $n = \overline{\mathtt{b_7b_6b_5b_4b_3b_2b_1b_0}}_{\ 2}$. Les valeurs de $b_0, \ldots, b_7$ sont inconnues.

Question 1 Donnez une expression utilisant exclusivement n et des opérateurs logiques qui ait pour résultat $\overline{\mathtt{b_4b_3}}_{\ 2}$.

Question 2 Donnez une expression utilisant exclusivement n et des opérateurs logiques qui ait pour résultat $\overline{\mathtt{b_7b_0}}_{\ 2}$.

Question 3 Donnez une expression utilisant exclusivement n et des opérateurs logiques qui ait pour résultat $\overline{\mathtt{b_7b_5111b_2}}_{\ 2}$.

1.7.6 Représentation des caractères

pour approfondir **Exercice 1-19** *Télécharger un nombre de Mersenne*

Question 1 Le plus grand entier évoqué à l'exercice 7 est stocké [5] dans un fichier texte de 23,7 Mo. Comment expliquez-vous la différence avec le nombre de Mo nécessaires au stockage de ce nombre ?

Exercice 1-20

En utilisant la vue sur un fichier texte depuis un éditeur hexadécimal (cf figure 1.2), remplissez le tableau suivant :

Car	bin	oct	déc	hexa
L				
				6c
			32	
		105		
	1100010			

Exercice 1-21 *Halloween et Noël*

Pourquoi les informaticiens confondent-ils Halloween et Noël ?

5. Sur ce site : `https://www.mersenne.org/primes/?press=M77232917`

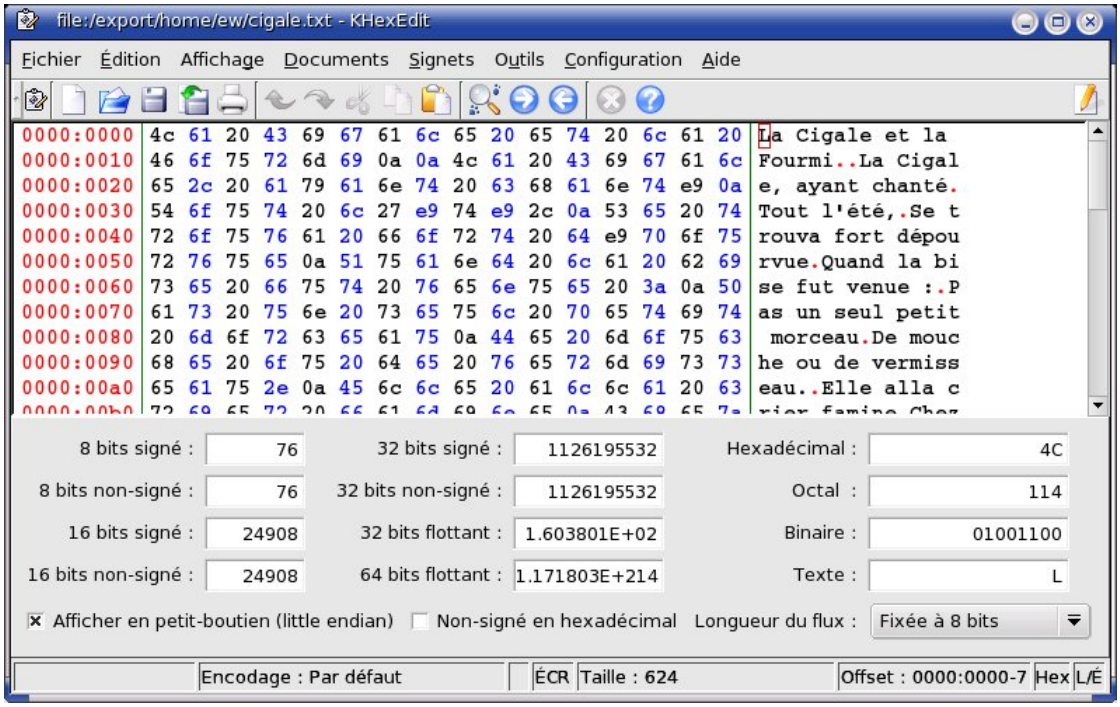


FIGURE 1.2 – Vue hexadécimale sur un fichier texte

Chapitre 2

Codes et codages

2.1 Exemples de codages

2.1.1 Le codage Morse

Le codage Morse, plus communément code Morse, a été mis au point en 1838 par Samuel Morse pour la télégraphie.

Il est d'usage d'utiliser deux symboles, le point · et le tiret −, pour représenter ce codage (cf table B.1 page 91). Mais en réalité, il faut rajouter un symbole, non graphiquement représenté : la coupure entre les codes, sans quoi ce codage ne serait pas décodable (cf page 37).

Ce codage permet de coder les 26 lettres de l'alphabet latin, les 10 chiffres décimaux, ainsi que certains symboles de ponctuation et autres caractères accentués spécifiques à certaines langues.

2.1.2 Le codage Baudot

Le codage Baudot est sans doute le premier codage binaire destiné à être utilisé par une machine. Il a été créé à l'origine par Émile Baudot en 1874 et a été ensuite modifié pour devenir le code CCITT n°2 qui sera utilisé par le Telex.

Ce codage permet de coder une soixantaine de caractères avec des mots binaires de 5 bits. (Cf table B.2 page 93).

Il peut sembler étrange de coder plus de 32 caractères avec des mots de 5 bits. Mais en fait, l'alphabet codé est découpé en deux, et deux mots de 5 bits particuliers (**11011** et **11111**) permettent de changer d'alphabet.

2.1.3 Le codage ASCII

Le codage ASCII (American Standard Code for Information Interchange) a été initialement conçu en 1963, puis modifié en 1967 pour inclure les lettres minuscules. Il est devenu la norme ISO 646 en 1983.

Ce codage code 128 caractères incluant les 26 lettres latines en version majuscule et minuscule, les 10 chiffres décimaux, l'espace, les symboles de ponctuation et de parenthésage. Les 32 premiers caractères sont des codes de formatage : retour chariot (CR), passage à la ligne (LF), tabulation horizontale (HT), ... ou de communication : début d'en-tête de transmission (SOH), début de texte transmis (STX), ... (Cf table B.3 page 94).

Ce codage ne code aucune version accentuée des lettres latines utilisée dans les langues européennes autre que l'anglais, et n'inclut encore moins les caractères spécifiques à d'autres alphabets : arabe, cyrillique, grec, ...

Dans le codage ASCII, tous les codes sont des mots binaires de 7 bits. Ils ont souvent été complétés d'un huitième bit :

— soit un bit de parité pour pouvoir détecter certaines erreurs pouvant survenir dans les communications (cf chapitre 4) ;

— soit un bit fixé à 0, les codes ASCII correspondant alors aux nombres de 0 à 127 codés en binaire.

2.1.4 Les codages ISO-8859

À la fin des années 1980, sont définis les codages ISO-8859 qui sont diverses extensions du codage ASCII permettant de coder en plus des lettres latines accentuées et d'introduire (en partie) certains autres alphabets (arabe, cyrillique, grec, hébreu).

Tous ces codages codent les caractères avec des mots binaires de 8 bits.

2.1.5 Le codage UTF-8

Dans les années 1990, le projet Unicode de codage de tous les alphabets est né. Unicode est une famille de codages, dont UTF-8 est un membre.

Le codage UTF-8 est un codage de longueur variable. Certains caractères sont codés sur un seul octet, ce sont les 128 caractères du codage ASCII. Le bit de poids fort des codes de ces caractères est toujours un `0`.

Les autres caractères peuvent être codés sur 2, 3 ou 4 octets. Les bits de poids fort du premier octet codant l'un de ces caractères indique toujours le nombre d'octets codant. Si le caractère est codé sur 2 octets, les trois premiers bits du premier octet sont `110`. S'il est codé sur 3 octets, alors les quatre premiers bits du premier octet sont `1110`. Et enfin s'il est codé sur 4 octets les cinq premiers bits du premier octet sont `11110`. Les deux bits de poids fort des octets qui suivent le premier octet codant sont toujours `10`. Voir la table 2.1 pour un résumé.

Nbre octets codant	Format du code
1	`0xxxxxxx`
2	`110xxxxx 10xxxxxx`
3	`1110xxxx 10xxxxxx 10xxxxxx`
4	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`

TABLE 2.1 – Format du codage UTF-8

2.2 Alphabets, mots et langages

Avant de donner la définition générale des codes et codages, nous allons préciser les notions d'*alphabet*, de *mot* et de *langage*.

2.2.1 Alphabets

Un *alphabet* est un ensemble fini non vide dont les éléments sont appelés selon les contextes *symboles*, *caractères* ou *lettres*. Nous désignerons les alphabets par des lettres majuscules de la

forme $\mathcal{A}$, $\mathcal{B}$ ...

Exemple 1 :
Selon les besoins les alphabets peuvent être très divers.

— Alphabet des 26 lettres latines (non accentuées) :

$$\mathcal{A} = \{\mathtt{A}, \mathtt{B}, \mathtt{C}, \ldots, \mathtt{Z}\}.$$

— Alphabet des symboles de la numération romaine :

$$\mathcal{A} = \{\mathtt{I}, \mathtt{V}, \mathtt{X}, \mathtt{L}, \mathtt{C}, \mathtt{D}, \mathtt{M}\}.$$

— Alphabet des chiffres de la numération décimale :

$$\mathcal{A} = \{\mathtt{0}, \mathtt{1}, \mathtt{2}, \ldots, \mathtt{9}\}.$$

— Alphabet de la numération binaire :

$$\mathcal{A} = \{\mathtt{0}, \mathtt{1}\}.$$

Remarque : Rien n'interdit de concevoir un alphabet qui ne contient qu'un seul symbole, mais un tel alphabet représente un intérêt extrêmement limité. Dans toute la suite, sauf mention explicite du contraire, les alphabets que nous considérerons seront constitués d'au moins deux symboles.

Nous noterons en général q le nombre de symboles qu'un alphabet contient.

$$\mathrm{card}(\mathcal{A}) = q.$$

2.2.2 Mots

Un *mot sur un alphabet* $\mathcal{A}$ est une suite finie éventuellement vide de symboles de $\mathcal{A}$. La *longueur* d'un mot est le nombre de symboles dans cette suite. Nous désignerons souvent les mots par des lettres grasses de la forme **u**, **v**, **w** ..., et la longueur d'un mot sera notée entre deux barres verticales (|).

$$|\mathbf{u}| = \text{longueur du mot } \mathbf{u}.$$

Exemple 2 :
Voici quelques exemples de mots correspondant aux alphabets de l'exemple 1.

— `CODAGE`, `TIMOLEON`, `ZTAGHUR`, `G` sont quatre mots sur l'alphabet latin de longueurs respectives 6, 8, 9 et 1. Le troisième exemple montre qu'un mot peut être **n'importe quelle** suite de symboles et pas seulement un mot figurant dans un dictionnaire.

— `IV`, `CLII` et `IIDVLM` sont trois mots sur l'alphabet de la numération romaine de longueurs repectives 2, 4 et 6. Encore une fois le troisième exemple montre qu'un mot peut être n'importe quelle suite de symboles.

Nous utiliserons les notations

1. ε pour désigner le mot vide ;
2. $\mathcal{A}^n$, $n \in \mathbb{N}$, pour désigner l'ensemble de tous les mots de longueur n sur l'alphabet $\mathcal{A}$;

3. $\mathcal{A}^*$ pour désigner l'ensemble de tous les mots sur l'alphabet $\mathcal{A}$:

$$\mathcal{A}^* = \bigcup_{n \geq 0} \mathcal{A}^n \ ;$$

4. $\mathcal{A}^+$ pour désigner l'ensemble de tous les mots non vides sur l'alphabet $\mathcal{A}$:

$$\mathcal{A}^+ = \bigcup_{n > 0} \mathcal{A}^n.$$

2.2.3 Concaténation de mots

La *concaténation* de deux mots est une opération qui à partir de deux mots $\mathbf{u}$ et $\mathbf{v}$ construit un troisième mot $\mathbf{w}$ dont les symboles sont ceux de $\mathbf{u}$ suivis de ceux de $\mathbf{v}$. Ainsi la longueur du mot $\mathbf{w}$ est la somme des longueurs des deux mots $\mathbf{u}$ et $\mathbf{v}$.

Exemple 3 :

Avec l'alphabet latin, les mots $\mathbf{u} = \texttt{CO}$ et $\mathbf{v} = \texttt{DAGE}$ donnent par concaténation

$$\mathbf{u}.\mathbf{v} = \texttt{CODAGE},$$

et

$$\mathbf{v}.\mathbf{u} = \texttt{DAGECO}.$$

Propriétés de la concaténation

— Associativité : pour tous mots $\mathbf{u}$, $\mathbf{v}$ et $\mathbf{w}$, on a

$$(\mathbf{u}.\mathbf{v}).\mathbf{w} = \mathbf{u}.(\mathbf{v}.\mathbf{w}).$$

Cette propriété assure la non ambiguïté du non usage des parenthèses lorsqu'on écrit $\mathbf{u}.\mathbf{v}.\mathbf{w}$ pour la concaténation de trois mots.

— Élément neutre : le mot vide laisse tout mot inchangé par concaténation. Pour tout mot $\mathbf{u}$, on a

$$\varepsilon.\mathbf{u} = \mathbf{u}.\varepsilon = \mathbf{u}.$$

— Non commutativité comme le montre l'exemple 3. En général

$$\mathbf{u}.\mathbf{v} \neq \mathbf{v}.\mathbf{u}.$$

Puissance n-ème d'un mot : Nous utiliserons la notation puissance pour désigner la concaténation d'un mot avec lui-même plusieurs fois.

$$\mathbf{u}^n = \underbrace{\mathbf{u}.\mathbf{u}.\ldots.\mathbf{u}}_{n \text{ fois}}.$$

Lorsque $n = 0$,

$$\mathbf{u}^0 = \varepsilon.$$

2.2.4 Préfixes d'un mot

Étant donnés deux mots $\mathbf{u}$ et $\mathbf{v}$ sur un même alphabet $\mathcal{A}$, on dit que $\mathbf{u}$ *est un préfixe* (ou plus simplement *est préfixe*) de $\mathbf{v}$ s'il existe un troisième mot $\mathbf{w} \in \mathcal{A}^*$ tel que

$$\mathbf{v} = \mathbf{u}.\mathbf{w}.$$

Autrement dit, si les premières lettres de $\mathbf{v}$ sont toutes les lettres de $\mathbf{u}$.

Exemple 4 :

— Le mot vide est préfixe de tous les mots. En effet on a vu que pour tout mot $\mathbf{u}$,

$$\mathbf{u} = \varepsilon.\mathbf{u}.$$

— Un mot est toujours préfixe de lui même. En effet, on a

$$\mathbf{u} = \mathbf{u}.\varepsilon.$$

— Les préfixes du mot `CODAGE` sont les mots ε, `C`, `CO`, `COD`, `CODA`, `CODAG` et `CODAGE`.

Nous noterons $\mathrm{Pref}(\mathbf{u})$ l'ensemble des préfixes d'un mot $\mathbf{u}$.

$$\mathrm{Pref}(\texttt{10010}) = \{\varepsilon, \texttt{1}, \texttt{10}, \texttt{100}, \texttt{1001}, \texttt{10010}\}.$$

Le nombre de préfixes d'un mot est égal à la longueur de ce mot plus un.

$$\mathrm{card}(\mathrm{Pref}(\mathbf{u})) = |\mathbf{u}| + 1.$$

2.2.5 Langages

On appelle *langage* sur un alphabet $\mathcal{A}$ tout sous-ensemble de $\mathcal{A}^*$.

Exemple 5 :

Pour n'importe quel alphabet $\mathcal{A}$, $\emptyset$ et $\mathcal{A}^*$ sont deux langages sur $\mathcal{A}$. Le premier ne contient aucun mot, et le second contient tous les mots de $\mathcal{A}^*$: ce sont, pour un alphabet donné, le plus petit et le plus gros langage.

Voici d'autres exemples de langages.

— Le langage des mots construits sur l'alphabet latin qui figure dans l'édition 2010 du Petit Robert. Ce langage contient environ 60 000 mots.

— Le langage des mots construits sur l'alphabet binaire dont la première lettre est un `1`, et est suivie de n'importe quel nombre de `0`. Ce langage contient une infinité de mots.

— $\mathcal{A}^n$, n étant un entier naturel, est le langage des mots de longueur n sur l'alphabet $\mathcal{A}$. C'est un langage qui contient q^n mots si l'alphabet contient q symboles.

Remarque : Comme le montre l'exemple 5, les langages peuvent être infinis ou non. Dans ce cours nous condidèrerons essentiellement des langages finis.

2.2.6 Concaténation de langages

On peut étendre la notion de concaténation des mots à celle des langages. Étant donnés deux langages L_1 et L_2 sur le même alphabet $\mathcal{A}$, on appelle *langage concaténé* de L_1 et L_2, noté $L_1.L_2$, le langage constitué des mots de $\mathcal{A}^*$ qui peuvent être obtenus par concaténation d'un mot de L_1 et un mot de L_2.

$$L_1.L_2 = \{\mathbf{u} \in \mathcal{A} \mid \exists \mathbf{u_1} \in L_1, \mathbf{u_2} \in L_2 \text{ tq } \mathbf{u} = \mathbf{u_1}.\mathbf{u_2}\}.$$

Exemple 6 :

— Avec l'alphabet latin, si $L_1 = \{\texttt{CO}, \texttt{TIM}\}$ et $L_2 = \{\texttt{DAGE}, \texttt{OLEON}\}$, alors

$$L_1.L_2 = \{\texttt{CODAGE}, \texttt{COOLEON}, \texttt{TIMDAGE}, \texttt{TIMOLEON}\}.$$

— Avec l'alphabet binaire, si $L_1 = \{\texttt{0}, \texttt{01}\}$ et $L_2 = \{\texttt{1}, \texttt{11}\}$, alors

$$L_1.L_2 = \{\texttt{01}, \texttt{011}, \texttt{0111}\}.$$

Le dernier exemple ci-dessus montre que contrairement à ce qu'on pourrait être amené à croire, le nombre de mots dans un langage concaténé de deux langages finis n'est pas égal au produit des nombres de mots de chacun des deux langages. Mais, on a tout de même la relation suivante

Proposition 2.1. *Si L_1 et L_2 sont deux langages finis, alors*

$$\text{card}(L_1.L_2) \leq \text{card}(L_1)\text{card}(L_2).$$

Puissance n-ème d'un langage : Comme pour les mots, on utilise la notation puissance pour désigner un langage concaténé avec lui-même n fois.

$$L^n = \underbrace{L.L.\ldots.L}_{n \text{ fois}}.$$

Lorsque $n = 0$,

$$L^0 = \{\varepsilon\}.$$

Étoile d'un langage : L'étoile d'un langage L, notée L^*, est l'ensemble de tous les mots que l'on peut former en concaténant des mots de L. Formellement, c'est le langage obtenu par réunion de toutes les puissances de L.

$$L^* = \bigcup_{n \geq 0} L^n.$$

2.2.7 Représentation arborescente d'un langage

Il est parfois pratique d'avoir une représentation graphique des langages sous forme d'arbres. Les figures 2.1 et 2.2, par exemple, montrent l'ensemble des mots sur l'alphabet binaire $\{\texttt{0}, \texttt{1}\}$ pour la première, et l'ensemble des mots sur l'alphabet ternaire $\{\texttt{0}, \texttt{1}, \texttt{2}\}$ pour la seconde. Dans les deux cas, la représentation est un *arbre* dont la racine est le mot vide, racine à partir de

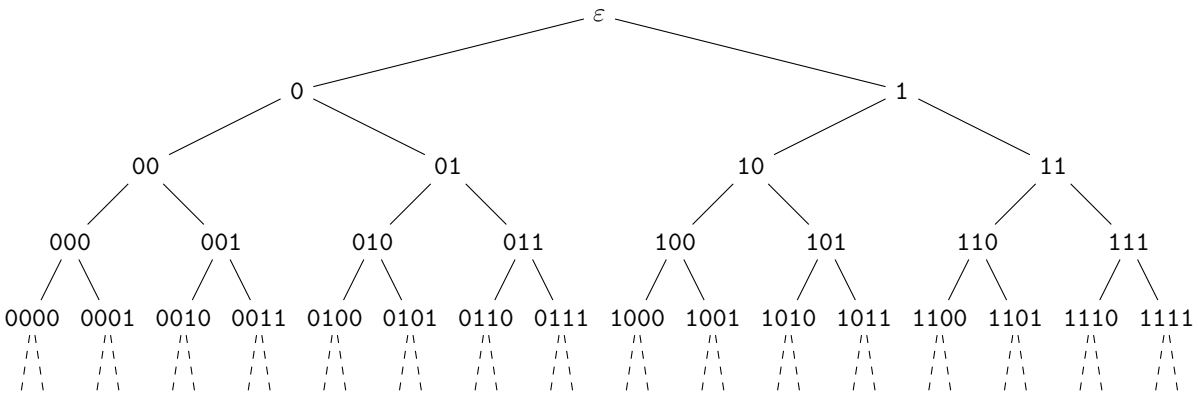


FIGURE 2.1 – Arbre des mots binaires (limités à la longueur 4)

laquelle on peut joindre n'importe quel mot le long d'une branche. Dans le cas binaire, à partir de n'importe quel mot partent deux arcs, et dans le cas ternaire il y en a trois.

Pour représenter graphiquement un langage, nous utiliserons cette représentation arborescente, mais nous distiguerons les mots du langage de ceux qui n'en font pas partie en marquant les premiers d'un grisé comme ceci : mot du langage. La figure 2.3 montre l'arbre du langage binaire

$$L = \{\mathtt{00}, \mathtt{01}, \mathtt{110}, \mathtt{001}\}.$$

Il faut remarquer qu'on trouve les préfixes d'un mot sur la branche qui sépare le mot vide du mot lui-même. On reviendra sur cette remarque plus tard à propos des langages préfixes.

2.3 Codes et codages

Nous allons maintenant donner un sens précis aux deux mots *code* et *codage*. Nous verrons que dans ce cours, le sens que nous leur donnons n'est pas toujours le sens usuel. En particulier, nous verrons que contrairement à l'usage le morse n'est pas un code mais un codage.

Avant de fixer ce vocabulaire, il est nécessaire de discuter de la factorisation d'un mot dans un langage.

2.3.1 Factorisation d'un mot dans un langage

La factorisation des mots dans un langage est un peu l'analogue de la factorisation des nombres entiers en nombres premiers. Il est connu que tout nombre entier au moins égal à un se décompose d'une seule façon (à l'ordre près des facteurs) en un produit de nombres premiers. Pour les mots, la question qui se pose est de savoir si un mot peut être obtenu par concaténation de plusieurs mots d'un langage fixé.

Plus précisément, étant donnés un langage $L \subset \mathcal{A}^*$ et un mot $\mathbf{u} \in \mathcal{A}^*$, existe-t-il des mots $\mathbf{u_1}$, $\mathbf{u_2}$, ..., $\mathbf{u_p}$ dans L tels que

$$\mathbf{u} = \mathbf{u_1}.\mathbf{u_2}.\ldots.\mathbf{u_p} \ ?$$

Dans l'affirmative, on dira que $\mathbf{u}$ est *factorisable dans* L, et que les mots $\mathbf{u_1}$, $\mathbf{u_2}$, ..., $\mathbf{u_p}$ (dans cet ordre) forment une *factorisation* de $\mathbf{u}$ dans L. Autrement dit, $\mathbf{u}$ est factorisable dans L si et

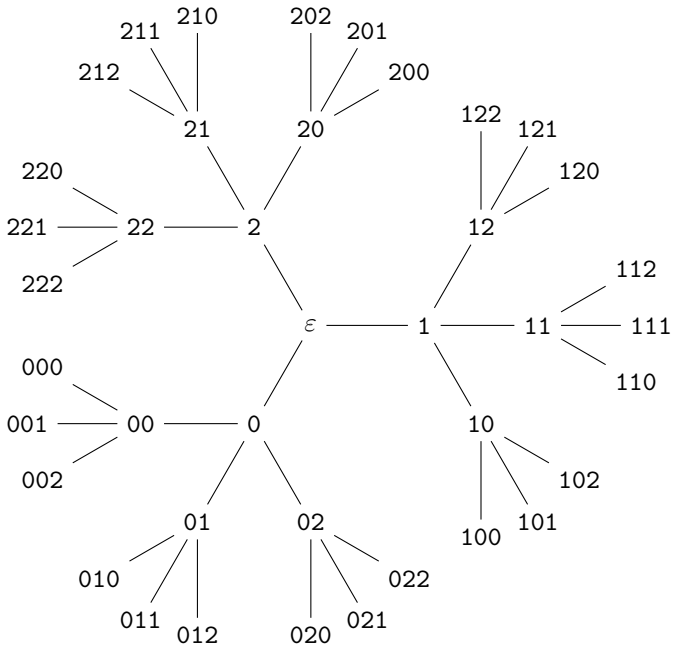


FIGURE 2.2 – Arbre des mots ternaires (limités à la longueur 3)

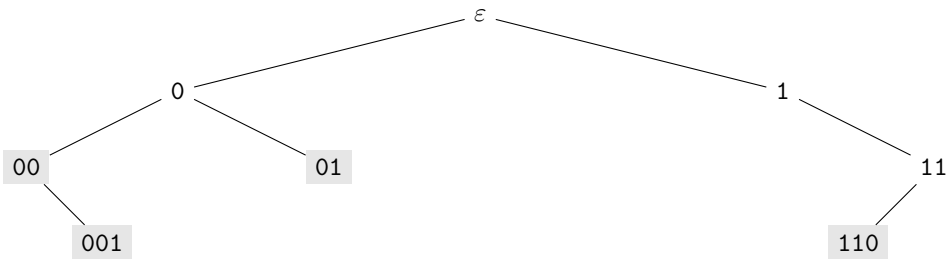


FIGURE 2.3 – Arbres des mots du langage $L = \{00, 01, 110, 001\}$

seulement si $\mathbf{u} \in L^*$.

Exemple 7 :

Sur l'alphabet $\mathcal{A} = \{0, 1\}$, avec le langage $L = \{0, 10, 01\}$.

— Le mot $\mathbf{u} = 00110$ est factorisable dans L. En effet, $\mathbf{u}$ peut se décomposer en

$$\mathbf{u} = 0.01.10,$$

et chaque mot de cette décomposition est dans L. De plus, cette décomposition est unique.

— Le mot $\mathbf{v} = 010$ est factorisable dans L de deux façons :

$$\mathbf{v} = 0.10 = 01.0.$$

— Le mot $\mathbf{w} = 1100$ n'est pas factorisable dans L, car il est impossible d'obtenir un mot débutant par deux 1 consécutifs en concaténant des mots de L.

Exemple 8 :

Pour un alphabet $\mathcal{A}$ quelconque, si on prend $L = \mathcal{A}$, alors tout mot possède une factorisation unique dans L. Il suffit de prendre les mots d'une lettre pour la factorisation, et c'est évidemment la seule façon d'obtenir une factorisation.

2.3.2 Codes

Un *code* est un langage dans lequel tous les mots ne possèdent au plus qu'une seule factorisation.

Commençons par donner des exemples de langages qui ne sont pas des codes.

Exemple 9 :

— Le langage de l'exemple 7 n'est pas un code, puisqu'il existe des mots ayant plusieurs factorisations dans ce code.

— Un langage contenant le mot vide n'est pas un code, puisque dans un tel langage tout mot ayant une factorisation dans L admet plusieurs factorisations dans L obtenues par insertion du mot vide dans la factorisation initiale. En effet si $\mathbf{u_1}$, $\mathbf{u_2}$, ..., $\mathbf{u_p}$ est une factorisation d'un mot $\mathbf{u}$ avec des mots de L, alors on obtient (une infinité) d'autres factorisations de $\mathbf{u}$ par

$$\begin{aligned}\mathbf{u} &= \varepsilon.\mathbf{u_1}.\mathbf{u_2}....\mathbf{u_p}\\ &= \mathbf{u_1}.\varepsilon.\mathbf{u_2}....\mathbf{u_p}\\ &= \mathbf{u_1}.\mathbf{u_2}.\varepsilon...\mathbf{u_p}\\ &= \dots\end{aligned}$$

En particulier, le mot vide se factorise en

$$\begin{aligned}\varepsilon &= \varepsilon\\ &= \varepsilon^2\\ &= \varepsilon^3\\ &= \dots\end{aligned}$$

Donnons maintenant quelques exemples de codes.

Exemple 10 :

— Lorsque $L = \mathcal{A}$, L est un code, puisque tout mot de $\mathcal{A}^*$ se factorise de manière unique en des mots d'une lettre.

— Le langage $L = \{\mathtt{0}, \mathtt{10}\}$ (obtenu en supprimant l'un des mots du langage de l'exemple 7) est un code. En effet, on peut distinguer trois sortes de mots (les deux premières sortes n'étant pas exclusives l'une de l'autre)

1. les mots se terminant par `1` qui ne peuvent pas se factoriser dans L ;
2. les mots contenant deux `1` consécutifs qui ne se factorisent pas dans L ;
3. les autres mots, qui ne contiennent donc pas deux `1` consécutifs et qui se terminent par `0`, qui peuvent tous se factoriser d'une seule façon avec les deux mots de L.

— Le langage $L = \{\texttt{10}^n \mid n \in \mathbb{N}\}$, langage contenant tous les mots débutant par `1` suivi d'un nombre (n) quelconque de `0`, est un code. Tous les mots binaires débutant par `1` possèdent une factorisation unique dans L, les autres n'en ont pas.
— Le langage de la figure 2.3 est un code.

Proposition 2.2. *Si C est un code fini, alors pour tout entier $n \in \mathbb{N}$*

$$\text{card}(C^n) = \text{card}(C)^n.$$

Démonstration. Laissée en exercice. □

Remarque : Contrairement aux nombres premiers qui permettent d'engendrer tous les nombres entiers par multiplication, en général un code ne permet pas de composer tous les mots par concaténation. Il peut exister des mots n'ayant pas de factorisation (cf exemple 10).

2.3.3 Codages

Intuitivement, un codage transforme des mots construits sur un alphabet source $\mathcal{S}$ en des mots sur un alphabet cible $\mathcal{A}$. Le codage ASCII transforme tout symbole d'un alphabet qui en contient 128 en un mot binaire de 8 bits. Les codages ISO-LATIN transforment tous les symboles d'un alphabet qui en contient 224 en un mot binaire de longueur 8. Le codage UTF-8 transforme les symboles d'un alphabet qui en contient plus de deux millions en un mot binaire de 8, 16, 24 ou 32 bits.

Formellement, un *codage* est une application

$$\mathbf{c} \ : \mathcal{S}^* \longrightarrow \mathcal{A}^*$$

satisfaisant les trois propriétés suivantes :

1. l'application $\mathbf{c}$ est injective, c.-à-d. deux mots différents ont des codages différents

$$\forall \mathbf{u}, \mathbf{v} \in \mathcal{S}^*, \mathbf{u} \neq \mathbf{v} \Rightarrow \mathbf{c}(\mathbf{u}) \neq \mathbf{c}(\mathbf{v}).$$

2. l'application $\mathbf{c}$ est compatible avec l'opération de concaténation.

$$\forall \mathbf{u}, \mathbf{v} \in \mathcal{S}^*, \mathbf{c}(\mathbf{u}.\mathbf{v}) = \mathbf{c}(\mathbf{u}).\mathbf{c}(\mathbf{v}).$$

3. le mot vide est transformé en lui même

$$\mathbf{c}(\varepsilon) = \varepsilon.$$

La première de ces trois propriétés est nécessaire sans quoi il serait impossible de décoder certains mots. La seconde et la troisième propriété sont des propriétés qui rendent effective la procédure d'encodage (cf algorithmes 2.1 et 2.2) et qui permettent la caractérisation donnée par le théorème suivant.

Definition 1. *Caractérisation d'un codage*

Un codage $\mathbf{c} \ : \mathcal{S}^* \longrightarrow \mathcal{A}^*$ est entièrement caractérisé par les mots associés aux symboles de $\mathcal{S}$.

De plus, le langage

$$C = \{\mathbf{c}(x) \mid x \in \mathcal{S}\}$$

est un code qui sera appelé *code associé* au codage.

Remarque : Dans cette définition d'un codage nous exigeons que la fonction **c** soit injective pour des raisons évidentes liées à la nécessité de pouvoir *décoder* une information codée. Cela implique en particulier que le nombre de mots du code associé soit au moins égal à celui du nombre de symboles de l'alphabet source

$$\text{card}(\mathcal{S}) \leq \text{card}(C).$$

Si le nombre de mots du code est strictement plus grand que le nombre de symboles de l'alphabet à coder, il y a des mots du code qui ne servent à rien. On peut donc supposer, sans perte de généralité, que ces deux nombres sont égaux

$$\text{card}(\mathcal{S}) = \text{card}(C),$$

et comme les alphabets que nous considérons sont des ensembles finis, cela explique le fait que nous nous intéressions principalement aux codes finis. La fonction de codage établit alors une bijection entre l'alphabet source et le code.

Autre remarque : La donnée d'un code seul ne suffit évidemment pas à caractériser le codage d'un alphabet. Il faut expliciter la correspondance entre chaque symbole de l'alphabet à coder et le mot du code qui lui est associé. C'est le rôle de la fonction **c**. Pour un alphabet et un code de cardinal q donnés, cela donne $q! = 1 \times 2 \times \ldots \times q$ codages possibles.

Exemple 11 :

- La table des caractères ASCII est un codage d'un alphabet de 128 symboles par des mots binaires de 7 bits (cf l'annexe B.3 pour avoir la correspondance complète). Le code est constitué des 128 mots binaires de longueur 8 dont le bit de poids fort est 0.
- Les codages ISO-8859 constituent bien des codages au sens où nous venons de le définir. Pour le codage ISO-8859-1 (ou ISO-LATIN1), l'alphabet source est constitué de 224 caractères (les 128 caractères de l'alphabet ASCII, auxquels viennent s'ajouter 96 autres caractères), et le code sous-jacent est formé de mots binaires de longueur 8 (cf l'annexe B.4).
- Toute permutation des lettres d'un alphabet, c'est-à-dire toute fonction bijective qui à une lettre d'un alphabet associe une lettre du même alphabet, est un codage. C'est par exemple de tels codages qui ont été employés par le passé pour rendre confidentiels certains messages [1]. Ce domaine des codages est celui d'une discipline voisine qu'on nomme *cryptographie*.

Algorithmes d'encodage : Les algorithmes 2.1 et 2.2 sont des algorithmes, récursif pour le premier et itératif pour le second, d'encodage d'un mot quelconque par un codage quelconque. Dans ces deux algorithmes, le codage d'une lettre ($\mathbf{c}(x)$) peut être obtenu par consultation d'une table de codage.

Décodage : L'opération inverse de l'encodage est le *décodage*. Mathématiquement parlant, c'est la fonction réciproque de celle du codage. Si le codage transforme des mots sur l'alphabet $\mathcal{S}$ en des mots sur l'alphabet $\mathcal{A}$

$$\mathbf{c} \ : \ \mathcal{S}^* \to \mathcal{A}^*,$$

1. Par exemple, Jules César dissimulait le sens des messages destinés à ses généraux en décalant de trois rangs toutes les lettres de l'alphabet (latin). Le `A` devenant ainsi un `D`, le `B` un `E`, ... et le `Z` un `C`.

Algorithme 2.1 Algorithme récursif de codage

Entrée : une table de codage $\mathbf{c} : \mathcal{S} \to C$, et un mot $\mathbf{u} \in \mathcal{S}^*$.
Sortie : le mot $\mathbf{v} = \mathbf{c}(\mathbf{u})$.

```
si u = ε alors
    renvoyer ε
sinon {u est de la forme x.w, où x est une lettre}
    renvoyer c(x).codage(w)
fin si
```

Algorithme 2.2 Algorithme itératif de codage

Entrée : une table de codage $\mathbf{c} : \mathcal{S} \to C$, et un mot $\mathbf{u} \in \mathcal{S}^*$.
Sortie : le mot $\mathbf{v} = \mathbf{c}(\mathbf{u})$.

```
v := ε
pour chaque lettre x de u par un parcours de gauche à droite faire
    v := v.c(x)
fin pour
renvoyer v
```

le décodage doit permettre de retrouver un mot de $\mathcal{S}^*$ à partir du mot de $\mathcal{A}^*$ qui le code, d'où (en notant $\mathbf{d}$ la fonction de décodage)

$$\mathbf{d} : \mathcal{A}^* \to \mathcal{S}^*,$$

et on doit avoir pour tout mot $\mathbf{u} \in \mathcal{S}^*$

$$\mathbf{d}(\mathbf{c}(\mathbf{u})) = \mathbf{u},$$

autrement dit, si on décode un mot codé, on doit obtenir le mot initial.

Toutefois, on peut souligner deux différences entre les opérations de codage et de décodage.

1. Si la fonction de codage peut s'appliquer à tous les mots sur l'alphabet source, celle de décodage n'est pas nécessairement définie sur tous les mots de l'alphabet cible. En effet, il peut exister des mots de $\mathcal{A}^*$ n'ayant aucune factorisation dans le code sous-jacent au codage, et par conséquent non décodable. Par exemple, dans le codage UTF-8, le mot `11000011.00100001` n'est pas décodable (voyez-vous pourquoi ?).
2. La mise en oeuvre algorithmique du décodage est moins simple que celle du codage en général. Cependant, pour certaines familles de codes présentées dans la section qui suivent, le problème du décodage est relativement simple.

2.4 Quelques familles de codes

Trois familles de codes sont présentées pour lesquelles le problème du décodage est relativement simple.

2.4.1 Codes de longueur fixe

On dit d'un code qu'il est de *longueur fixe* lorsque tous les mots qu'il contient ont la même longueur. Autrement dit $C \subset \mathcal{A}^*$ est un code de longueur fixe s'il existe un entier non nul n tel

que

$$C \subseteq \mathcal{A}^n.$$

Exemple 12 :

— Les 128 mots (octets) utilisé pour le codage ASCII forment un code de longueur fixe.
— Le code associé à l'un quelconque des codages ISO-8859 est un code de longueur fixe.
— Le code associé à l'UTF-8 n'est pas un code de longueur fixe.

Il est facile de décoder les mots obtenus par un codage de longueur fixe. L'algorithme 2.3 décrit la procédure de décodage.

Algorithme 2.3 Décodage pour les codes de longueur fixe

Entrée : un code C de longueur fixe n, un codage $\mathbf{c} : \mathcal{S} \to C$, un mot $\mathbf{v} \in \mathcal{A}^*$
Sortie : le mot $\mathbf{u} \in \mathcal{S}^*$ tel que $\mathbf{c}(\mathbf{u}) = \mathbf{v}$ s'il existe.

si $|\mathbf{v}|$ n'est pas multiple de n **alors**
  renvoyer ECHEC
sinon
  Découper $\mathbf{v}$ en facteurs de longueurs n

$$\mathbf{v} = \mathbf{v_1}.\mathbf{v_2}.\ldots.\mathbf{v_p}.$$

fin si
pour chaque $\mathbf{v_i}$ **faire**
  chercher $x_i \in \mathcal{S}$ tel que $\mathbf{c}(x_i) = \mathbf{v_i}$
  si x_i n'existe pas **alors**
    renvoyer ECHEC
  fin si
fin pour
renvoyer $\mathbf{u} = x_1.x_2.\ldots.x_p.$

2.4.2 Codes « à virgule »

Les *codes à virgule* sont des codes pour lesquels un symbole (ou un groupe de symboles) de $\mathcal{A}$ est réservé pour marquer le début ou la fin des mots du code. Ce symbole particulier est appelé une *virgule*. Il permet de factoriser un message codé avec ce code.

Exemple 13 :

— Le Morse est un code à virgule. En effet, contrairement à ce qu'on pourrait croire, l'alphabet cible n'est pas limité aux deux seuls symboles point · et tiret ▬, ce qui le rendrait indécodable. En effet, le mot ·▬· possède deux factorisations : (·▬).(·) correspondant à `AE` et (·).(▬·) correspondant à `EN`. Le Morse est un code sonore dans lequel les silences servent à séparer les mots du code. Le silence est donc un symbole particulier du code, la virgule, qui marque la fin de tous les mots.
— Le troisième code de l'exemple 10 page 33 est un code à virgule, le symbole `1` fait office de virgule.

Le décodage pour les codes à virgule est décrit par l'algorithme 2.4 page suivante.

Algorithme 2.4 Décodage pour les codes à virgule

Entrée : un code C à virgule, un codage $\mathbf{c} : \mathcal{S} \to C$, un mot $\mathbf{v} \in \mathcal{A}^*$
Sortie : le mot $\mathbf{u} \in \mathcal{S}^*$ tel que $\mathbf{c}(\mathbf{u}) = \mathbf{v}$ s'il existe.

```
Découper v en mots v_1, v_2, ..., v_p en cherchant les virgules dans v.
pour chaque v_i faire
    chercher x_i ∈ S tel que c(x_i) = v_i
    si x_i n'existe pas alors
        renvoyer ECHEC
    fin si
fin pour
renvoyer u = x_1.x_2.....x_p.
```

2.4.3 Codes préfixes

La famille des codes préfixes est une famille très riche pour laquelle le problème du décodage est relativement simple. Cette famille contient celle des codes de longueur fixe, et aussi tous les codes à virgule dans lesquels la virgule est placée à la fin des mots.

Un *langage préfixe* est un langage dans lequel aucun mot n'est le préfixe d'un autre mot du langage. Dit autrement, $L \subset \mathcal{A}^*$ est préfixe si pour tous mots $\mathbf{u}, \mathbf{v} \in L$, on a

$$\mathbf{u} \notin \mathrm{Pref}(\mathbf{v}) \text{ et } \mathbf{v} \notin \mathrm{Pref}(\mathbf{u}).$$

Exemple 14 :

- Le langage $L = \{\mathtt{00}, \mathtt{010}, \mathtt{10}, \mathtt{110}, \mathtt{111}\}$ est préfixe.
- Les langages de longueur fixe sont des langages préfixes.
- Les langages à virgule dont la virgule est placée en fin des mots codants sont des langages préfixes. C'est le cas du Morse.
- Le code UTF-8 est un langage préfixe.
- Le langage vide $L = \emptyset$ est un langage préfixe.
- Le langage $L = \{\varepsilon\}$ est un langage préfixe, et c'est le seul langage préfixe contenant le mot vide.

Propriétés des langages préfixes :

Théorème 2.1. *Tout langage préfixe autre que $\{\varepsilon\}$ est un code.*

Preuve du théorème 2.1. Nous allons démontrer le théorème par contraposition, c'est-à-dire en montrant qu'un langage qui n'est pas un code n'est pas un langage préfixe.

Soit donc $L \subseteq \mathcal{A}^*$ un langage non réduit au seul mot vide qui n'est pas un code, et supposons qu'il existe des mots admettant deux factorisations différentes dans L. Parmi ces mots, choisissons en un de longueur minimale que nous noterons $\mathbf{u}$. Il existe donc deux entiers p et r (éventuellement égaux) et des mots $\mathbf{u_1}$, $\mathbf{u_2}$, ..., $\mathbf{u_p}$ et $\mathbf{u'_1}$, $\mathbf{u'_2}$, ..., $\mathbf{u'_r}$ dans L tels que

$$\begin{aligned} \mathbf{u} &= \mathbf{u_1}.\mathbf{u_2}.\dots.\mathbf{u_p} \\ &= \mathbf{u'_1}.\mathbf{u'_2}.\dots.\mathbf{u'_r}, \end{aligned}$$

autrement dit,

$$\mathbf{u_1}.\mathbf{u_2}.\dots.\mathbf{u_p} = \mathbf{u'_1}.\mathbf{u'_2}.\dots.\mathbf{u'_r} \tag{2.1}$$

Comme on a supposé que le mot $\mathbf{u}$ est de longueur minimale parmi ceux ayant une double factorisation dans L, les deux mots $\mathbf{u_1}$ et $\mathbf{u'_1}$ ont des longueurs différentes, car dans le cas contraire l'équation 2.1 montre que ces deux mots sont égaux, et se simplifie en

$$\mathbf{u_2}.\ldots.\mathbf{u_p} = \mathbf{u'_2}.\ldots.\mathbf{u'_r},$$

et cela contredirait le caractère minimal de la longueur de $\mathbf{u}$.

Enfin quitte à échanger leurs rôles, on peut supposer que

$$|\mathbf{u_1}| < |\mathbf{u'_1}|.$$

En tenant compte de cette inégalité des longueurs, et de l'équation 2.1, on en déduit que $\mathbf{u_1}$ est un préfixe de $\mathbf{u'_1}$. Et comme ces deux mots sont des mots de L, on en déduit que L n'est pas préfixe, ce que nous voulions démontrer.

□

Remarque : La réciproque du théorème 2.1 n'est pas vraie. Il existe des langages non préfixes qui sont des codes. C'est par exemple le cas du langage $L = \{\mathtt{10}^n \mid n \in \mathbb{N}\}$ donné dans l'exemple 10 page 33. C'est aussi le cas du langage

$$L = \{\mathtt{00}, \mathtt{01}, \mathtt{110}, \mathtt{001}\}$$

qui n'est pas un langage préfixe (car `00` est préfixe de `001`), mais néanmoins est un code comme nous le verrons dans la section 2.6.1 page 42.

Arbre d'un code préfixe : L'arbre d'un code préfixe possède la particularité que le long d'une branche issue de la racine, il ne peut y avoir qu'un seul mot du langage. Dans l'arbre d'un code préfixe, les branches peuvent être délimitées par les mots du code. Ce n'est pas le cas pour les langages non préfixes comme celui mentionné plus haut (cf figure 2.3 page 32).

La figure 2.4 montre l'arbre du premier code donné dans l'exemple 14 page ci-contre. On constate que cet arbre est limité à cinq branches, chacune d'elles terminée par l'un des cinq mots de ce code.

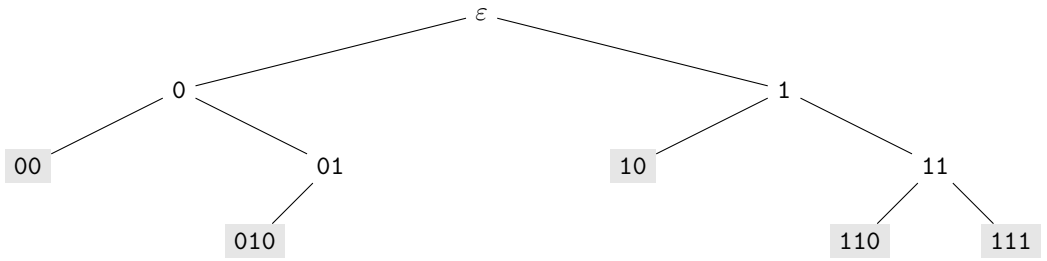


FIGURE 2.4 – Représentation arborescente d'un code préfixe

2.5 Existence de codes

La question que nous allons traiter dans cette section est celle de l'existence ou non d'un code contenant un nombre donné de mots dont les longueurs sont elles aussi données. Plus précisément, étant donnés

Algorithme 2.5 `decode_lettre(v,a)` : Décodage récursif d'une lettre pour un codage préfixe

Entrée : Un sous-arbre non vide a d'un codage préfixe $\mathbf{c} : \mathcal{S} \to \mathcal{A}^*$ et un mot $\mathbf{v} \in \mathcal{A}^*$ non vide.
Sortie : Le couple $(x, \mathbf{v}') \in \mathcal{S} \times \mathcal{A}^*$ tel que $\mathbf{v} = \mathbf{c}(x).\mathbf{v}'$ s'il existe.

si a est une feuille **alors**
  renvoyer le couple (x,$\mathbf{v}$) où x est la lettre attachée à a
sinon
  soit y la première lettre de $\mathbf{v}$ et $\mathbf{v}'$ tel que $\mathbf{v} = y.\mathbf{v}'$
  si le sous-arbre de a correspondant à y est vide **alors**
    renvoyer ECHEC
  sinon
    soit a' le sous-arbre correspondant à y
    renvoyer decode_lettre($\mathbf{v}', a'$)
  fin si
fin si

Algorithme 2.6 Décodage d'un codage préfixe

Entrée : Un codage préfixe $\mathbf{c} : \mathcal{S} \to \mathcal{A}^*$ représenté par son arbre a, et un mot $\mathbf{v} \in \mathcal{A}^*$.
Sortie : Le mot $\mathbf{u} \in \mathcal{S}^*$ tel que $\mathbf{c}(\mathbf{u}) = \mathbf{v}$ s'il existe.

$\mathbf{u} := \varepsilon$
tant que $\mathbf{v} \neq \varepsilon$ **faire**
  $(x, \mathbf{v}) :=$ decode_lettre($\mathbf{v}, a$)
  $\mathbf{u} := \mathbf{u}$.x
fin tant que
renvoyer $\mathbf{u}$

1. un alphabet $\mathcal{A}$ à q symboles,
2. un entier $m \in \mathbb{N}$,
3. et m entiers l_1, l_2, ... et l_m,

existe-t-il un code $C \subset \mathcal{A}^*$ contenant m mots dont les longueurs sont l_1, l_2, ...l_m ?

Cette question apparaît naturellement lorsqu'on se demande s'il est possible de coder les symboles d'un alphabet $\mathcal{S}$ avec des mots sur un alphabet $\mathcal{A}$ dont les longueurs sont imposées.

Nous allons traiter d'abord cette question pour les codes de longueur fixe pour lesquelles la réponse est simple, puis nous allons répondre pour les codes préfixes, pour enfin terminer avec les codes quelconques.

Dans toute la suite, nous appellerons *distribution de longueurs* d'un langage fini la donnée des longueurs l_1, l_2, $\ldots l_m$ des mots de ce langage, et nous la noterons $[l_1, l_2, \ldots, l_m]$.

2.5.1 Existence de codes de longueur fixe

Dans le cas où toutes les longueurs sont égales à un certain entier n non nul,

$$l_1 = l_2 = \ldots = l_m = n,$$

la réponse à la question est positive si et seulement si $m \leq q^n$.

2.5.2 Existence de codes préfixes

Théorème 2.2. Théorème de Kraft

Soit $[l_1, l_2, \ldots, l_m]$ une distribution de longueurs, et $\mathcal{A}$ un alphabet contenant q symboles. Il existe un code préfixe $C \subset \mathcal{A}^*$ contenant un mot pour chacune de ces longueurs si et seulement si

$$\sum_{k=1}^{m} q^{-l_k} \leq 1.$$

2.5.3 Existence de codes quelconques

Pour les codes quelconques la réponse est identique à celle des codes préfixes. C'est le théorème de Mc Millan que nous admettrons.

Théorème 2.3. Théorème de Mc Millan

Soit $[l_1, l_2, \ldots, l_m]$ une distribution de longueurs, et $\mathcal{A}$ un alphabet contenant q symboles. Il existe un code $C \subset \mathcal{A}^*$ contenant un mot pour chacune de ces longueurs si et seulement si

$$\sum_{k=1}^{m} q^{-l_k} \leq 1.$$

La seule différence entre les deux théorèmes réside dans l'absence du qualificatif préfixe pour le code dont une condition nécessaire et suffisante d'existence est donnée.

Voici un corollaire important du théorème de McMillan.

Corollaire 2.1. *Pour tout code fini, il existe un code préfixe ayant la même distribution de longueurs.*

Preuve du corollaire 2.1. Soit C un code et $[l_1, \ldots, l_m]$ la distribution des longueurs de ce code. D'après le théorème de Mc Millan, on a

$$\sum_{k=1}^{m} q^{-l_k} \leq 1,$$

qui est une condition suffisante d'existence d'un code préfixe de même distribution des longueurs (théorème de Kraft). □

Ce corollaire montre donc qu'on ne perd rien à vouloir se limiter aux seuls codes préfixes.

On appelle *somme de Kraft* d'un langage fini L le nombre réel

$$K(L) = \sum_{\mathbf{u} \in L} q^{-|\mathbf{u}|}.$$

Si $[l_1, \ldots, l_m]$ est une distribution des longueurs de L on a aussi

$$K(L) = \sum_{i=1}^{m} q^{-l_i}.$$

Pour un langage fixé, le théorème 2.3 donne une condition nécessaire, mais pas suffisante pour que ce langage soit un code.

Corollaire 2.2. *Soit $L \subset \mathcal{A}^*$ un langage.*
Si L est un code fini, alors sa somme de Kraft est inférieure ou égale à 1.

$$K(L) \leq 1.$$

Cette condition n'est pas suffisante comme le montre l'exemple $L = \{\mathtt{0}, \mathtt{00}\}$ qui n'est pas un code, mais dont la somme de Kraft vaut 3/4. Il faut donc bien faire attention. La somme de Kraft ne permet pas d'affirmer qu'un langage particulier est un code. Elle permet seulement d'affirmer que la distribution des longueurs des mots d'un langage est compatible avec un code. Dans l'exemple ci-dessus, ce que permet de savoir la somme de Kraft est qu'il existe un code avec un mot de longueur 1 et un mot de longueur 2.

2.6 Algorithme de décision pour un code

Pour terminer ce chapitre, nous allons voir qu'il est possible de décider si un langage est un code ou non. La question est donc : étant donné un langage $L \subset \mathcal{A}^*$, ce langage est-il un code ou non ? existe-t-il un algorithme permettant de répondre à cette question ?

Nous nous limiterons à la question pour les langages finis.

La réponse est évidente pour les langages préfixes. Ce sont tous des codes (sauf $\{\varepsilon\}$). Qu'en est-il pour un langage fini non préfixe ?

2.6.1 Un exemple « à la main »

Revenons sur un exemple de langage non préfixe cité page 39 comme étant un code

$$L = \{\mathtt{00}, \mathtt{01}, \mathtt{110}, \mathtt{001}\}.$$

Nous allons prouver qu'il est effectivement un code.

Si L n'est pas un code, il existe un mot $\mathbf{u} \in \mathcal{A}^*$ ($\mathcal{A} = \{0, 1\}$ ici) qui possède deux factorisations distinctes avec des mots de L

$$\begin{aligned}\mathbf{u} &= \mathbf{u_1}.\mathbf{u_2}.\ldots.\mathbf{u_p} \\ &= \mathbf{u'_1}.\mathbf{u'_2}.\ldots.\mathbf{u'_r},\end{aligned}$$

où p et r sont deux entiers (éventuellement égaux) et $\mathbf{u_1}$, $\mathbf{u_2}$, ..., $\mathbf{u_p}$ et $\mathbf{u'_1}$, $\mathbf{u'_2}$, ..., $\mathbf{u'_r}$ sont des mots de L.

Comme déjà mentionné dans la preuve du théorème 2.1, on peut toujours supposer que

$$|\mathbf{u_1}| < |\mathbf{u'_1}|,$$

et on en déduit alors que $\mathbf{u_1}$ est un préfixe de $\mathbf{u'_1}$.

En cherchant les candidats possibles dans L on trouve $\mathbf{u_1} = 00$ et $\mathbf{u'_1} = 001$ comme seule possibilité. En éliminant les lettres de $\mathbf{u_1}$, la double factorisation se résume maintenant à

$$\mathbf{u_2}.\ldots.\mathbf{u_p} = 1.\mathbf{u'_2}.\ldots.\mathbf{u'_r}, \tag{2.2}$$

équation de laquelle on déduit que le mot $\mathbf{u_2}$ doit commencer par 1.

Un seul mot de L commence par 1, on en déduit que $\mathbf{u_2} = 110$. En éliminant la lettre initiale de $\mathbf{u_2}$ dans les deux membres de (2.2), on obtient

$$10.\ldots.\mathbf{u_p} = \mathbf{u'_2}.\ldots.\mathbf{u'_r}, \tag{2.3}$$

ce qui montre que $\mathbf{u'_2}$ doit avoir 10 comme préfixe. Or aucun mot de L n'admet ce préfixe.

Cela contredit l'existence d'un mot ayant deux factorisations dans L et établit donc le fait que L est un code non préfixe.

En général la vérification qu'un langage est un code peut s'avérer plus fastidieuse. Les notions de résiduels et de quotients de langages permettent de systématiser la procédure suivie ci-dessus, et aboutissent à un algorithme de décision pour le problème envisagé ici.

2.6.2 Quotient à gauche d'un mot

Le *quotient à gauche d'un mot* $\mathbf{u}$ par un autre mot $\mathbf{v}$ définis sur $\mathcal{A}^*$ est le mot $\mathbf{w}$ tel que $\mathbf{u} \cdot \mathbf{w} = \mathbf{v}$. Ce quotient à gauche est noté

$$\mathbf{u}^{-1} \cdot \mathbf{v} = \mathbf{w}$$

Si $\mathbf{u}$ n'est pas préfixe de $\mathbf{v}$, alors le quotient à gauche de $\mathbf{u}$ par $\mathbf{v}$ n'est pas défini.

2.6.3 Résiduel d'un langage

Le *résiduel à gauche* d'un langage $L \subseteq \mathcal{A}^*$ par un mot $\mathbf{u} \in \mathcal{A}^*$ est le langage que l'on obtient en ne gardant des mots de L que ceux admettant $\mathbf{u}$ comme préfixe, et en supprimant ce préfixe de ces mots. En notant $\mathbf{u}^{-1}.L$ ce résiduel, on a

$$\mathbf{u}^{-1}.L = \{\mathbf{w} \in \mathcal{A}^* \mid \mathbf{u}.\mathbf{w} = \mathbf{v} \in L\}.$$

On peut aussi définir de manière analogue la notion de *résiduel à droite* par

$$L.\mathbf{u}^{-1} = \{\mathbf{w} \in \mathcal{A}^* \mid \mathbf{w}.\mathbf{u} = \mathbf{v} \in L\},$$

mais nous n'utiliserons pas cette notion et donc par la suite nous dirons plus simplement *résiduel* sans préciser *à gauche* qui sera sous-entendu.

Exemple 15 :
Avec le langage $L = \{\mathtt{00}, \mathtt{01}, \mathtt{110}, \mathtt{001}\}$
— et avec le mot $\mathbf{u_1} = \mathtt{10}$, on a
$$\mathbf{u_1}^{-1}.L = \emptyset$$
puisqu'aucun mot de L n'admet $\mathbf{u_1}$ pour préfixe ;
— et avec le mot $\mathbf{u_2} = \mathtt{0}$, on a
$$\mathbf{u_2}^{-1}.L = \{\mathtt{0}, \mathtt{1}, \mathtt{01}\}$$
puisque trois mots de L ($\mathtt{00}$, $\mathtt{01}$, $\mathtt{001}$) admettent $\mathbf{u_2}$ pour préfixe ;
— et avec le mot $\mathbf{u_3} = \mathtt{00}$, on a
$$\mathbf{u_3}^{-1}.L = \{\varepsilon, \mathtt{1}\}$$
puisque deux mots de L ($\mathtt{00}$ et $\mathtt{001}$) admettent $\mathbf{u_3}$ pour préfixe.

Propriétés des résiduels : Pour un langage L et un mot $\mathbf{u}$, on a toujours les propriétés :
— $\varepsilon^{-1}.L = L$.
— $\varepsilon \in \mathbf{u}^{-1}.L \Leftrightarrow \mathbf{u} \in L$.
— $\mathbf{u}^{-1}.L = \emptyset \Leftrightarrow \mathbf{u} \notin \mathrm{Pref}(L)$.

2.6.4 Quotient d'un langage

Le *quotient à gauche* d'un langage $L \subseteq \mathcal{A}^*$ par un langage $M \subseteq \mathcal{A}^*$ est le langage obtenu en ne gardant que les mots de L ayant un préfixe dans M et en supprimant ce préfixe des mots gardés. En notant $M^{-1}.L$ ce quotient, on a

$$M^{-1}.L = \{\mathbf{v} \in \mathcal{A}^* \mid \exists \mathbf{u} \in M \text{ tq } \mathbf{u}.\mathbf{v} \in L\}.$$

Autrement dit, le quotient à gauche est la réunion de tous les résiduels à gauche de L par les mots de M
$$M^{-1}.L = \bigcup_{\mathbf{u} \in M} \mathbf{u}^{-1}.L.$$

De manière similaire, on peut définir le *quotient à droite* d'un langage par un autre, mais n'utilisant pas cette dernière notion, nous dirons plus simplement *quotient* sans autre précision pour parler des quotients à gauche.

Exemple 16 :
Avec le langage L et les trois mots $\mathbf{u_1}$, $\mathbf{u_2}$ et $\mathbf{u_3}$ de l'exemple 15, en posant $M = \{\mathbf{u_1}, \mathbf{u_2}, \mathbf{u_3}\}$ on a
$$M^{-1}.L = \{\varepsilon, \mathtt{0}, \mathtt{1}, \mathtt{01}\}.$$

Propriétés des quotients : Pour tous langages L et M on a
— $\varepsilon \in M^{-1}.L \Leftrightarrow M \cap L \neq \emptyset$. En particulier, pour tout langage L non vide, $\varepsilon \in L^{-1}.L$.
— $M^{-1}.L = \emptyset \Leftrightarrow M \cap \mathrm{Pref}(L) = \emptyset$. En particulier, si L est un langage préfixe non vide, $L^{-1}.L = \{\varepsilon\}$.

2.6.5 Algorithme de Sardinas et Patterson

Soit $L \subset \mathcal{A}^*$ un langage. On définit à partir de L une suite (L_n) de langages en posant

$$L_0 = L \tag{2.4}$$
$$L_1 = L^{-1}.L \setminus \{\varepsilon\} \tag{2.5}$$

et pour tout $n \geq 1$

$$L_{n+1} = L^{-1}.L_n \cup L_n^{-1}.L \tag{2.6}$$

Théorème 2.4. *Un langage $L \subset \mathcal{A}^*$ est un code si et seulement si le mot vide n'est dans aucun des langages L_n de la suite définie ci-dessus.*

Exemple 17 :
Voici quelques de suites de langages L_n obtenues sur divers exemples.

1. Lorsque L est un langage de longueur fixe (non nulle), alors $L_1 = L^{-1}.L \setminus \{\varepsilon = \emptyset$, et par conséquent, pour tout entier $n \geq 2$, $L_n = \emptyset$. Le mot vide n'est dans aucun langage L_n, et L est bien évidemment un code.
2. Lorsque L est un langage préfixe (autre que le langage réduit au seul mot vide), $L_1 = \emptyset$ et il en est de même pour tous les langages L_n avec $n \geq 2$. Et tout langage préfixe est un code.
3. Pour le langage $L = \{\mathtt{1}, \mathtt{00}, \mathtt{01}, \mathtt{10}\}$, on a

   $$L_1 = \{\mathtt{0}\}$$
   $$L_2 = \{\mathtt{0}, \mathtt{1}\}$$
   $$L_3 = \{\mathtt{0}, \mathtt{1}, \varepsilon\}$$

   Comme $\varepsilon \in L_3$, L n'est pas un code. La raison pour laquelle $\varepsilon \in L_3$ réside dans le fait que $\mathtt{1} \in L$ et $\mathtt{1} \in L_2$ ($1^{-1}.1 = \varepsilon$). Et la raison pour laquelle $\mathtt{1} \in L_2$ provient du fait que $1 = 0^{-1}.01$ avec $\mathtt{0} \in L_1$ et $\mathtt{01} \in L$. Enfin $\mathtt{0} \in L_1$ parce que $0 = 1^{-1}.10$ avec $\mathtt{1} \in L$ et $\mathtt{10} \in L$. Ainsi on a bien construit une double factorisation du mot $\mathtt{101}$ avec des mots de L :

   $$\mathtt{101} = \underbrace{\mathtt{1}}.\underbrace{\mathtt{01}} = \underbrace{\mathtt{10}}.\underbrace{\mathtt{1}}$$

4. Enfin pour le langage $L = \{\mathtt{000}, \mathtt{010}, \mathtt{011}, \mathtt{01001}\}$, on a

   $$L_1 = \{\mathtt{01}\}$$
   $$L_2 = \{\mathtt{0}, \mathtt{1}, \mathtt{001}\}$$
   $$L_3 = \{\mathtt{00}, \mathtt{10}, \mathtt{11}, \mathtt{1001}\}$$
   $$L_4 = \{\mathtt{0}\}$$
   $$L_5 = \{\mathtt{00}, \mathtt{10}, \mathtt{11}, \mathtt{1001}\}$$

   Comme $L_5 = L_3$, on en déduit que pour tout indice pair à partir de 4 on a

   $$L_{2n} = L_4,$$

   et tout indice impair à partir de 3,

   $$L_{2n+1} = L_3.$$

   Ainsi, le mot vide n'est dans aucun des langages L_n, et L est un code.

Pour les langages finis on a la propriété importante décrite dans le lemme qui suit.

Lemme 2.1. *Lorsque L est un langage fini, la suite (L_n) est périodique, et la période est donné par le premier indice n pour lequel il existe un indice $k < n$ tel que $L_k = L_n$.*

Preuve du lemme 2.1. Les langages L_n sont tous des langages inclus dans l'ensemble des suffixes des mots de L. Si L est fini, alors il en est de même, et il n'y a qu'un nombre fini d'ensembles possibles pour les L_n. On a donc nécessairement dans la suite de ces langages deux langages égaux.

Soient donc $k < n$ les deux plus petits indices pour lesquels $L_k = L_n$.

Puisque le calcul d'un langage de la suite ne fait intervenir que le langage qui précède et le langage initial L, on en déduit que

$$L_{n+1} = L_{k+1},$$

et plus généralement pour tout entier $i \geq 0$

$$L_{n+i} = L_{k+i}.$$

Ce qui montre que la suite est périodique, de période $n - k$ à partir de l'indice k. □

Le lemme 2.1 assure qu'en un nombre fini de calculs, on a déterminé tous les langages intervenant dans la suite (L_n), ce qui donne un algorithme pour décider si un langage est ou non un code. C'est l'algorithme 2.7 connu sous le nom d'algorithme de Sardinas et Patterson.

Algorithme 2.7 Algorithme de Sardinas et Patterson

Entrée : L un langage.
Sortie : renvoie `vrai` si L est un code, et `faux` sinon.

```
L_0 := L, L_1 := L^{-1}.L \ {ε}.
n := 1.
tant que ε ∉ L_n et L_n ≠ L_k pour tout 0 ≤ k < n faire
  n := n + 1.
  L_n := L^{-1}.L_{n-1} ∪ L_{n-1}^{-1}.L.
fin tant que
si ε ∈ L_n alors
  renvoyer faux
sinon
  renvoyer vrai
fin si
```

2.7 Exercices

2.7.1 Première série

Codages ASCII, ISO-8859-1 et UTF-8

Exercice 2-1 *Avec le codage ASCII*

Pour cet exercice, utilisez la table ASCII de la page 94

Question 1 Codez votre nom en hexadécimal avec le codage ASCII.

Question 2 Voici le début d'un fichier texte vu avec un éditeur hexadécimal.

```
00000000   4C 61 20 43  69 67 61 6C  65 20 65 74  20 6C 61 20
00000010   46 6F 75 72  6D 69 0A 0A  4C 61 20 43  69 67 61 6C
00000020   65 2C 20 61  79 61 6E 74  20 63 68 61  6E 74 65 0A
00000030   54 6F 75 74  20 6C 27 65  74 65 2C 0A  53 65 20 74
```

Quel est ce texte ?

Question 3

Si on change le 3e bit de poids fort du second octet du fichier précédent, que devient le fichier texte ?

Exercice 2-2 *Avec le codage ISO-8859-1*

On rappelle que le codage ISO-8859-1 (encore appelé ISO-LATIN-1), est un codage de 224 caractères tous codés sur un octet (codes 0 à 127, et 160 à 255. Les 128 premiers caractères sont identiques aux caractères du codage ASCII, les 96 suivants contiennent (presque) tous les autres caractères utilisés en français comme par exemple les caractères accentués.

Voici une vue hexadécimale sur le début de la fable de La Fontaine « La cigale et la fourmi » codée en ISO-LATIN-1.

```
00000000   4C 61 20 43  69 67 61 6C  65 20 65 74  20 6C 61 20  La Cigale et la
00000010   46 6F 75 72  6D 69 0A 0A  4C 61 20 43  69 67 61 6C  Fourmi..La Cigal
00000020   65 2C 20 61  79 61 6E 74  20 63 68 61  6E 74 E9 0A  e, ayant chant..
00000030   54 6F 75 74  20 6C 27 E9  74 E9 2C 0A  53 65 20 74  Tout l'.t.,.Se t
```

Quel est le code du caractère `é` ?

Exercice 2-3 *Avec le codage UTF-8*

On rappelle que le codage UTF-8 est un codage de longueurs variables, dont les mots du code sont constitués de 1, 2, 3 ou 4 octets. La table 2.1 (page 26) donne le format des codes selon le nombre d'octets.

Question 1 Combien de caractères le codage UTF-8 permet-il de coder ?

Voici une vue hexadécimale sur le début de la fable de La Fontaine « La cigale et la fourmi » codée en UTF-8.

```
00000000   4C 61 20 43  69 67 61 6C  65 20 65 74  20 6C 61 20  La Cigale et la
00000010   46 6F 75 72  6D 69 0A 0A  4C 61 20 43  69 67 61 6C  Fourmi..La Cigal
00000020   65 2C 20 61  79 61 6E 74  20 63 68 61  6E 74 C3 A9  e, ayant chant..
00000030   0A 54 6F 75  74 20 6C 27  C3 A9 74 C3  A9 2C 0A 53  .Tout l'..t..,.S
```

Question 2 Quel est le code du caractère `é` ? Ce code a-t-il un rapport avec le code du même caractère dans le codage ISO-LATIN-1 ?

Exercice 2-4 *Conversion ISO-8859-1 vers UTF-8*

Question 1 Sur combien d'octets est codé un caractère avec le codage ISO-8859-1 ?

Question 2 En UTF-8, sur combien d'octets sont codés les caractères correspondants à ceux de l'ISO-8859-1 ?

Question 3 Prenons un caractère ISO-8859-1 qui est codé sur un seul octet en UTF-8. Comment convertir ce caractère en UTF-8 ?

Question 4 Prenons maintenant un caractère ISO-8859-1 qui est codé sur deux octets en UTF-8. Comment convertir ce caractère en UTF-8 en utilisant uniquement des opérations logiques ?

Exercice 2-5 *Avec un codage de longueur variable*

Dans cet exercice on considère le codage décrit par la table 2.2.

A 1010	B 0010011	C 01001	D 01110	E 110	F 0111100	G 0111110
H 0010010	I 1000	J 011111110	K 011111111001	L 0001	M 00101	N 1001
0 0000	P 01000	Q 0111101	R 0101	S 1011	T 0110	U 0011
V 001000	W 011111111000	X 01111110	Y 0111111111	Z 01111111101	ESP 111	

TABLE 2.2 – Codage de longueur variable

Question 1 Codez votre nom avec ce codage de longueur variable.

Question 2 Décodez le message `1000100101111000000010100101101001101000000001001`.

Question 3 Si on change le 3^{e} bit du message précédent, que devient ce message ? Même question en remplaçant 3^{e} par 2^{e}, puis en remplaçant 3^{e} par 12^{e}.

Exercice 2-6

À l'aide d'un éditeur hexadécimal, on visualise le contenu d'un fichier.

```
00000000  26 12 FE 03 50
```

Question 1 Sachant que ce fichier contient une information textuelle codée avec le codage de longueur variable (cf table 2.2), trouvez l'information codée.

Question 2 Quelle taille (en octets) aurait cette information si elle avait été codée à l'aide du code ASCII ?

Question 3 Quelle serait cette taille si on utilisait un codage de longueur fixe minimale ?

Codage d'images

pour approfondir **Exercice 2-7** *Codage* bitmap *des images*

Le codage *bitmap* d'une image consiste à coder la couleur de chaque pixel (point) de l'image par une chaîne binaire de longueur fixée. Ainsi, pour représenter une image ne contenant que du noir et du blanc, il suffit de coder chacune de ces deux couleurs sur un bit. Les formats les plus fréquents codent chaque pixel sur 1 bit (2 couleurs), 4 bits (16 couleurs), 8 bits (256 couleurs) et 24 bits ($\approx 16 \times 10^6$ couleurs).

Question 1

Une image Noir et Blanc de 10 lignes de 16 pixels est codée en bitmap. Voici la représentation hexadecimale de cette image

```
0000 0F00 1088 0388 0CBE 1088 1188 0E80 0000 7FFE
```

Dessinez l'image.

Question 2

Calculez la taille (en nombre d'octets) de la plus petite représentation possible en bitmap d'une image comprenant 200×300 pixels et 100 couleurs.

Question 3 *Images compressibles*

La représentation "bitmap" d'une image peut être inutilement gourmande en espace mémoire (Pensez à une image blanche).

Proposez une représentation plus compacte des images.

Exercice 2-8 *Espace mémoire d'un film non compressé*

Prenons un film HD muet (d'une résolution de 1920×1080 pixels à 25 images par seconde) d'une durée de 1h23m20s.

Question 1 Combien d'octets occuperait un tel film en supposant que chaque pixel est représenté par trois octets (un pour le rouge, le vert et le bleu) ?

2.7.2 Deuxième série

Exercice 2-9 *Nombre de mots*

Pour les deux questions qui suivent l'alphabet contient au moins les deux lettres `a` et `b`.

Question 1 Combien y a-t-il de mots de 5 lettres commençant par `a` et terminant par `b` ?

Question 2 Combien y a-t-il de mots de 10 lettres comprenant trois `a` et 2 `b` ?

Exercice 2-10 *Peut-on tout compresser ?*

Soit $\mathcal{A}$ un alphabet à q lettres.

Question 1 Combien y a-t-il de mots de longueur strictement inférieure à n ($n \neq 0$) ?

Un fichier est une suite finie d'octets. On peut considérer que c'est un mot sur l'alphabet $\mathcal{A}$ des 256 octets.

Un logiciel de compression transforme un fichier en un autre fichier qu'on espère être plus petit. Bien entendu une compression n'est utile que si l'opération inverse, la décompression, existe. On peut modéliser un logiciel de compression comme une fonction z de l'ensemble des mots dans lui-même :

$$z \ : \ \mathcal{A}^+ \to \mathcal{A}^*,$$

telle que pour tout mot $\mathbf{u} \in \mathcal{A}^+$, on a

$$|z(\mathbf{u})| < |\mathbf{u}|,$$

autrement dit le fichier obtenu par compression est strictement plus petit que le fichier d'origine.

Question 2 Quelle propriété mathématique la fonction z doit-elle posséder pour assurer l'existence de l'opéation de décompression ?

Question 3 En considérant la compression des fichiers de taille n, n étant un entier fixé, déduisez de la propriété trouvée pour z l'inexistence d'une telle fonction, et concluez.

Exercice 2-11 *Mots commutants*

Soit l'alphabet $\mathcal{A} = \{\mathtt{a}, \mathtt{b}\}$.

Question 1 Trouvez deux mots $\mathbf{u}$ et $\mathbf{v}$ non vides sur l'alphabet $\mathcal{A}$ tels que $\mathbf{u}.\mathbf{v} = \mathbf{v}.\mathbf{u}$.

Question 2 Montrez que pour tout mot $\mathbf{w} \in \mathcal{A}^*$ et tout entier $p, q \in \mathbb{N}$, les mots $\mathbf{u} = \mathbf{w}^p$ et $\mathbf{v} = \mathbf{w}^q$

commutent.

pour approfondir **Question 3** Réciproquement, montrez que la condition précédente est nécessaire pour que deux mots commutent.

Exercice 2-12 *Concaténation de langages*

Soient L et M deux langages sur un alphabet $\mathcal{A}$. La *concaténation* de L et M est le langage noté $L.M$ défini par

$$L.M = \{\mathbf{u}.\mathbf{v} \mid \mathbf{u} \in L,\ \mathbf{v} \in M\}$$

autrement dit, $L.M$ est l'ensemble de tous les mots obtenus en concaténant un mot de L à un mot de M.

On considère l'alphabet $\mathcal{A} = \{\mathtt{a}, \mathtt{b}\}$.

Question 1 Calculez le langage $L.M$ dans chacun des cas suivants

1. $L = \{\varepsilon, \mathtt{ab}, \mathtt{b}, \mathtt{aba}\}$, $M = \{\mathtt{a}, \mathtt{ab}\}$
2. $L = \{\mathtt{a}, \mathtt{ab}, \mathtt{bba}\}$, $M = \{\mathtt{a}, \mathtt{ba}, \mathtt{aab}\}$
3. $L = \emptyset$, M quelconque.

Question 2 À quelle condition nécessaire et suffisante, a-t-on $L \subseteq L.M$? et $M \subseteq L.M$?

Question 3 Lorsque L et M sont deux langages finis, quel est le nombre maximal de mots de $L.M$? Dans quel cas ce maximum est-il atteint ?

2.7.3 Troisième série

Exercice 2-13 *Code ou pas code ?*

Parmi les langages suivants sur l'alphabet $\mathcal{A} = \{\mathtt{0}, \mathtt{1}\}$ indiquez ceux qui sont des codes.

1. $L = \emptyset$
2. $L = \{\varepsilon\}$
3. $L = \{\mathbf{u}\}$, où $\mathbf{u}$ est un mot non vide sur l'alphabet $\mathcal{A}$
4. $L = \{\mathtt{0011}, \mathtt{1001}, \mathtt{0110}\}$
5. $L = \{\mathtt{00}, \mathtt{010}, \mathtt{111}, \mathtt{01101}, \mathtt{1100}\}$
6. $L = \{\mathtt{010101}, \mathtt{0101}\}$
7. $L = \{\mathtt{0}, \mathtt{01}, \mathtt{110}, \mathtt{1101}, \mathtt{1111}\}$
8. $L = \{\mathtt{0}, \mathtt{01}, \mathtt{100}\}$

Exercice 2-14

Question 1 Peut-on trouver un sous-ensemble strict de $Y = \{\mathtt{0}, \mathtt{01}, \mathtt{110}, \mathtt{1101}, \mathtt{1111}\}$ qui ne soit pas un code ?

Question 2 Peut-on trouver un code contenant strictement Y ?

Question 3 Reprendre les deux questions avec $Y = \{\mathtt{0}, \mathtt{01}, \mathtt{100}\}$.

Exercice 2-15

Quels sont les codes sur un alphabet à une seule lettre ?

Exercice 2-16

Soient u et v deux mots sur un alphabet $\mathcal{A}$. À quelle condition le langage constitué des deux mots u et v est-il un code ?

pour approfondir **Exercice 2-17**

Pour quelles valeurs des entiers naturels m, n et p, le langage

$$L = \{a^m, baba, ba^n, ab, bab^p\}$$

est-il un code ?

Exercice 2-18

Soit $C \subset \mathcal{A}^*$ un code fini contenant k mots, et $n \in \mathbb{N}$ un entier.

Question 1

Quel est le cardinal de C^n ?

Question 2

Quel est le cardinal de $C^{\leq n} = \bigcup_{0 \leq i \leq n} C^i$?

2.7.4 Quatrième série

Exercice 2-19

Les langages ci-dessous sont-ils des codes ? Si non, existe-t-il un code ayant le même alphabet et la même distribution de longueurs de mots ?

1. $L_1 = \{0, 01, 101, 110\}$
2. $L_2 = \{0, 01, 101, 110, 111\}$
3. $L_3 = \{0, 01, 102, 020, 211\}$

pour approfondir **Exercice 2-20** *Codage binaire de l'alphabet latin*

$\mathcal{S}$ désigne l'alphabet latin (26 lettres) et $\mathcal{A} = \{0, 1\}$.
On veut coder chaque lettre de $\mathcal{S}$ par un mot de $\mathcal{A}^*$

Question 1 On choisit un codage de longueur fixe. Quelle est la longueur minimale des mots du code ?

Question 2 On choisit l'application définie par $\mathbf{c}(a) = 0$, $\mathbf{c}(b) = 00$, $\mathbf{c}(c) = 000$, ... Cette application est-elle un codage ?

Existe-t-il un codage préfixe dont les mots ont même longueur que ceux de $\mathbf{c}$?

Question 3 Existe-t-il un codage préfixe possédant
— 8 mots de code de longueur 4,
— 8 mots de code de longueur 5,
— 8 mots de code de longueur 6,
— et 2 mots de code de longueur 7 ?

Question 4 Peut-on remplacer les mots de longueur 7 par des mots plus courts tout en gardant le caractère préfixe du codage et sans changer les autres mots ?

Question 5 Existe-t-il un codage préfixe à 8 mots de longueur 4 et 18 de longueur 5 ?

Exercice 2-21

Parmi les mots de code utilisés par un codage binaire préfixe, on trouve 0, 10 et 1100. Combien de mots de code de longueur 6 ce codage peut-il avoir ? Donnez un exemple d'un tel code préfixe.

pour approfondir

Exercice 2-22

On désire coder, à l'aide d'un alphabet $\mathcal{A} = \{0, 1, 2\}$, un alphabet $\mathcal{S}$ de dix lettres avec un mot-code d'une lettre, les neuf autres ayant trois lettres au maximum. On désire de plus que ce codage soit préfixe.

Question 1 Cela est-il possible ?

Question 2 Combien de mots-code de trois lettres faut-il au minimum ?

Exercice 2-23

Question 1 Vérifier que l'on peut raccourcir la longueur d'un mot d'un code binaire si et seulement si l'inégalité de Kraft est stricte.

Question 2 Qu'en est-il pour des codes sur un alphabet à plus de deux lettres ?

Exercice 2-24 *Codes maximaux*

Un code C est dit *maximal* si pour tout mot $u \in \mathcal{A}^*$, $C \cup \{u\}$ n'est pas un code.

Question 1 Montrez qu'un code préfixe fini est maximal si et seulement si $K(C) = 1$.

Question 2 Qu'en est-il pour un code sur un alphabet dont le cardinal $q > 2$?

Chapitre 3

Codages optimaux

3.1 Source

3.1.1 Source d'information

Une *source d'information* est caractérisée par

1. un alphabet de source $\mathcal{S} = \{s_1, s_2, \ldots, s_m\}$ décrivant quels sont les symboles s_i que cette source peut émettre ;
2. une distribution de probabilité $\mathrm{P} = \{p_1, p_2, \ldots, p_m\}$, chaque p_i étant un réel compris entre 0 et 1 donnant la probabilité que la source délivre le symbole $s_i \in \mathcal{S}$ et telle que $\sum_{i=0}^{m} p_i = 1$.

Plus formellement, une source est une variable aléatoire S prenant ses valeurs dans un ensemble fini $\mathcal{S}$ et dont la loi de probabilité est donnée par P. Ainsi pour tout symbole $s_i \in \mathcal{S}$

$$\Pr(S = s_i) = p_i.$$

Dans la suite, nous noterons $(\mathcal{S}, \mathrm{P})$ les sources d'information, c'est-à-dire sous la forme du couple (alphabet, distirbution de probabilité).

Exemple 1 :

— Les caractères utilisés dans différentes langues. L'alphabet peut être le même, mais les distributions de probabilité diffèrent (par exemple le caractère le plus fréquent en français est le e alors qu'en turc il s'agit du a).
— Les caractères utilisés dans différents langages de programmation. L'alphabet est toujours le même, mais les distributions de probabilité diffèrent.
— Source uniforme : source pour laquelle la distribution de probabilité est uniforme, c'est-à-dire pour laquelle la probabilité de chaque symbole est la même, ou autrement dit, pour tout symbole $s \in \mathcal{S}$, on a

$$\Pr(S = s) = \frac{1}{m}.$$

— Les octets successifs d'un fichier. L'alphabet source est l'ensemble des 256 octets, $\mathcal{S} = [\![0, 255]\!]$. La distribution de probabilités est définie par le nombre d'occurrences de chacun des octets. Si un octet s apparaît n_s fois dans le fichier, la probabilité associée est $p_s = \frac{n_s}{N}$, où N désigne le nombre d'octets de ce fichier.

3.1.2 Quantité d'information

La *quantité d'information* contenue dans un symbole $s \in \mathcal{S}$ d'une source $S = (\mathcal{S}, \mathrm{P})$, de probabilité non nulle est

$$I(s) = -\log_2 \Pr(S = s).$$

Par extension, pour un symbole s de probabilité nulle, on pose

$$I(s) = +\infty.$$

L'unité de la quantité d'information est le bit. Il s'agit du nombre de bits minimum nécessaire pour transmettre un symbole s. C'est une quantité théorique : le nombre minimum de bits n'est donc pas nécessairement un nombre entier (cela peut être 2,13 bits).

Propriétés :

1. Pour toute source S et tout symbole s de cette source, on a
$$0 \leq I(s).$$
2. De plus, $I(s) = 0$ si et seulement si $\Pr(S = s) = 1$, autrement dit si la source émet le symbole s avec certitude, et jamais les autres symboles.
3. La quantité d'information dans un symbole est infinie si et seulement si la probabilité de ce symbole est nulle, autrement dit si ce symbole n'est jamais produit par la source.
4. La quantité d'information contenue dans un symbole augmente lorsque la probabilité de ce symbole diminue.

Exemple 2 :

— Prenons un dé à 8 faces non pipé. L'information véhiculée par « Le dé a fait 5 » est de 3 bits.
— Prenons une pièce de monnaie biaisée qui tombe trois fois plus souvent sur pile que sur face ($P(pile) = 3/4$). La quantité d'information de « La pièce est tombée sur pile » est donc de $-\log_2(3/4) = \log_2(4/3) = \log_2(4) - \log_2(3)$. Ceci peut être vu comme le nombre de bits nécessaires pour représenter 4 valeurs distinctes ($\log_2(4)$) moins le nombre de bits nécessaires pour représenter 3 valeurs distinctes (puisque ce 3 représente en fait les 3 lancers tombant sur pile, qui véhiculent tous la même information).

3.1.3 Entropie d'une source

L'*entropie* d'une source S est la valeur moyenne, ou espérance mathématique, de la quantité d'information contenue dans chacun de ces symboles. On la note $H(S)$.

$$H(S) = \sum_{s \in \mathcal{S}} \Pr(S = s) I(s).$$

On peut aussi exprimer l'entropie par

$$\begin{aligned} H(S) &= \sum_{i=1}^{m} p_i I(s_i) \\ &= -\sum_{i=1}^{m} p_i \log_2 p_i \end{aligned}$$

En remarquant que $\lim_{x\to 0} x\log_2(x) = 0$, on peut poser par prolongement par continuité que $0\log_2(0) = 0$. Ainsi, la définition de l'entropie est valable lorsqu'une des probabilités p_i est nulle.

Exemple 3 :

1. Soit la source définie par l'alphabet $\mathcal{S} = \{s_1, s_2, s_3, s_4\}$ et la distribution de probabilité

s	s_1	s_2	s_3	s_4
$\Pr(S=s)$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$

   La quantité d'information contenue dans chacun des symboles est la même et vaut

$$I(s) = -\log_2 \frac{1}{4} = \log_2 4 = 2.$$

   L'entropie de la source est donc égale à

$$H(S) = 2.$$

2. Avec la distribution de probabilités

s	s_1	s_2	s_3	s_4
$\Pr(S=s)$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{8}$

   les quantités d'information contenues dans chacun des symboles sont

$$\begin{aligned} I(s_1) &= -\log_2 \frac{1}{2} = 1 \\ I(s_2) &= -\log_2 \frac{1}{4} = 2 \\ I(s_3) &= -\log_2 \frac{1}{8} = 3 \\ I(s_4) &= -\log_2 \frac{1}{8} = 3 \end{aligned}$$

   et l'entropie de la source vaut

$$H(S) = \frac{1}{2}\times 1 + \frac{1}{4}\times 2 + \frac{1}{8}\times 3 + \frac{1}{8}\times 3 = \frac{7}{4}.$$

3. Enfin avec la distribution de probabilités

s	s_1	s_2	s_3	s_4
$\Pr(S=s)$	1	0	0	0

   les quantités d'information contenues dans chacun des symboles sont

$$\begin{aligned} I(s_1) &= -\log_2 1 = 0 \\ I(s_2) &= +\infty \\ I(s_3) &= +\infty \\ I(s_4) &= +\infty \end{aligned}$$

   et l'entropie de la source vaut

$$H(S) = 1\times 0 + 0 + 0 + 0 = 0.$$

Propriétés :

1. L'entropie d'une source ne dépend que des valeurs p_i des probabilités des symboles et non des symboles eux-mêmes.
2. Pour toute source on a
$$0 \leq H(S) \leq \log_2 m.$$
3. L'entropie d'une source est nulle si et seulement si un symbole de la source a une probabilité égale à 1, les autres ayant tous une probabilité nulle.
4. L'entropie d'une source est maximale, c'est-à-dire égale à $\log_2 m$ si et seulement si la distribution de probabilité est uniforme.

L'unité de l'entropie d'une source est le bit.

3.2 Codages optimaux

Étant donnés une source $S = (\mathcal{S}, \mathrm{P})$ et un alphabet cible $\mathcal{A}$, on souhaite coder « *au mieux* » les symboles de $\mathcal{S}$ par des mots de $\mathcal{A}$.

3.2.1 Longueur moyenne d'un codage de source

La *longueur moyenne d'un codage* $\mathbf{c}$ est la moyenne des longueurs des mots utilisés dans le codage, longueurs coefficientées par la probabilité des symboles de source correspondant. Elle s'exprime par

$$\overline{n}_{\mathbf{c}} = \sum_{i=1}^{m} p_i |\mathbf{c}(s_i)|.$$

On peut aussi exprimer cette longueur moyenne comme étant l'espérance mathématique de la longueur des mots associés à chaque symbole de la source

$$\overline{n}_{\mathbf{c}} = \sum_{s \in \mathcal{S}} \Pr(S = s) |\mathbf{c}(s)|.$$

Exemple 4 :

Considérons à nouveau l'alphabet source $\mathcal{S} = \{s_1, s_2, s_3, s_4\}$, et les deux codages binaires des symboles de $\mathcal{S}$ définis par

s	s_1	s_2	s_3	s_4
$\mathbf{c_1}(s)$	00	01	10	11
$\mathbf{c_2}(s)$	0	11	100	101

Il est évident que la longueur moyenne du codage $\mathbf{c_1}$ est égale à 2 quelque soit la distribution de probabilités de la source.

$$\overline{n}_{\mathbf{c_1}} = 2.$$

Pour le second codage, sa longueur moyenne dépend de la distribution de probabilités. Voici ce qu'il en est avec les trois distributions envisagées dans l'exempe 3.

1. Avec la distribution uniforme

s	s_1	s_2	s_3	s_4
$\Pr(S=s)$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$

   la longueur moyenne est

$$\overline{n}_{\mathbf{c_2}} = \frac{1}{4}(1+2+3+3) = \frac{9}{4}.$$

2. Avec la distribution

s	s_1	s_2	s_3	s_4
$\Pr(S=s)$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{8}$

   la longueur moyenne est

$$\overline{n}_{\mathbf{c_2}} = \frac{1}{2}\times 1 + \frac{1}{4}\times 2 + \frac{1}{8}\times 3 + \frac{1}{8}\times 3 = \frac{7}{4}.$$

3. Et avec la distribution

s	s_1	s_2	s_3	s_4
$\Pr(S=s)$	1	0	0	0

   la longueur moyenne est

$$\overline{n}_{\mathbf{c_2}} = 1.$$

Propriétés :

1. La longueur moyenne d'un codage est toujours comprise entre les longueurs extrêmes des mots utilisés par le codage.

$$l \leq \overline{n}_{\mathbf{c}} \leq L$$

   si l (resp. L) est la longueur minimale (resp. maximale) d'un mot du code associé à $\mathbf{c}$.
2. Pour tout codage de longueur fixe égale à L, la longueur moyenne est égale à L :

$$\overline{n}_{\mathbf{c}} = L.$$

3. Pour toute source uniforme, la longueur moyenne de tout codage $\mathbf{c}$ est la moyenne arithmétique des longueurs des mots du code associé :

$$\overline{n}_{\mathbf{c}} = \frac{1}{m}\sum_{i=1}^{m} |\mathbf{c}(s_i)|.$$

4. Si avec un codage $\mathbf{c}$, on code une longue séquence de symboles de la source, on peut estimer la longueur du message ainsi codé à $\overline{n}_{\mathbf{c}} \times N$, où N est le nombre de symboles de source codés dans le message. Cette estimation est d'autant meilleure que N est grand.

Lorsque l'alphabet cible est l'alphabet binaire $\mathcal{A} = \{0, 1\}$, la longueur moyenne d'un codage s'exprime en bits par caractère.

3.2.2 Codage optimal d'une source

Comme les exemples précédents le montrent, les différents codages d'une même source n'ont pas la même longueur moyenne. Parmi tous les codages d'une source donnée, il en existe dont la longueur moyenne est minimale. De tels codages sont dits *optimaux*.

Un codage $\mathbf{c}$ d'une source $S = (\mathcal{S}, \mathrm{P})$ dans un alphabet cible $\mathcal{A}$ est dit *optimal*, si pour tout autre codage $\mathbf{c}'$ de la même source sur le même alphabet cible, on a

$$\overline{n}_{\mathbf{c}} \leq \overline{n}_{\mathbf{c}'}.$$

3.3 Théorème du codage sans bruit

Le théorème du codage sans bruit est dû à Claude Shannon (1948). Il donne une minoration de la longueur moyenne de tout codage d'une source, et une majoration de celle d'un codage optimal. Nous le présenterons en deux parties.

3.3.1 Première partie du théorème

Théorème 3.1. *Soit $S = (\mathcal{S}, \mathrm{P})$ une source.*
Pour tout codage $\mathbf{c}$ de S sur un alphabet cible à $q \geq 2$ éléments, on a

$$\frac{H(S)}{\log_2 q} \leq \overline{n}_{\mathbf{c}},$$

avec égalité si et seulement si pour tout i, on a $|\mathbf{c}(s_i)| = -\log_q p_i$.

Remarque : lorsque l'alphabet cible est l'alphabet binaire ($q = 2$), le théorème s'énonce
— Pour tout codage *binaire* $\mathbf{c}$ de S, on a

$$H(S) \leq \overline{n}_{\mathbf{c}},$$

avec égalité si et seulement si pour tout i, on a $|\mathbf{c}(s_i)| = -\log_2 p_i$.

3.3.2 Un lemme

Avant de prouver ce théorème nous allons d'abord démontrer le lemme suivant.

Lemme 3.1. *Soit $(p_1, p_2, \ldots, p_m)$ et $(q_1, q_2, \ldots, q_m)$ deux suites de m nombres réels positifs telles que*

$$\begin{aligned} \sum_{i=1}^{m} p_i &= 1 \\ \text{et } \sum_{i=1}^{m} q_i &\leq 1. \end{aligned}$$

Alors

$$\sum_{i=1}^{m} p_i \ln q_i \leq \sum_{i=1}^{m} p_i \ln p_i,$$

et on a l'égalité si et seulement si $p_i = q_i$ pour tout i.

Démonstration. La preuve s'appuie sur le fait que pour tout réel $x > 0$, on a

$$\ln x \leq x - 1,$$

avec égalité si et seulement si $x = 1$.
En effet, on a alors pour tout i

$$\ln \frac{q_i}{p_i} \leq \frac{q_i}{p_i} - 1,$$

avec égalité si et seulement si $p_i = q_i$.

En multipliant par p_i et en sommant sur i on obtient

$$\sum_{i=1}^{m} p_i \ln \frac{q_i}{p_i} \leq \sum_{i=1}^{m} q_i - \sum_{i=1}^{m} p_i,$$

avec égalité si et seulement si $p_i = q_i$ pour tout i.

Compte-tenu des hypothèses sur les deux suites (p_i) et (q_i), on en déduit l'inégalité

$$\sum_{i=1}^{m} p_i \ln \frac{q_i}{p_i} \leq 0.$$

Par conséquent

$$\sum_{i=1}^{m} p_i \ln q_i - \sum_{i=1}^{m} p_i \ln p_i \leq 0,$$

ce qui achève la preuve du lemme. □

3.3.3 Preuve du théorème 3.1

Soit donc $\mathbf{c}$ un codage de la source sur un alphabet cible à $q \geq 2$ éléments. Pour chaque entier $i \in [\![1, m]\!]$, désignons par $n_i = |\mathbf{c}(s_i)|$ la longueur du mot associé au symbole s_i, et posons

$$q_i = q^{-n_i}.$$

D'après le théorème de Mc Millan, comme $\mathbf{c}$ est un codage, on a

$$\sum_{i=1}^{m} q_i \leq 1.$$

En appliquant le lemme 3.1, on en déduit l'inégalité

$$\sum_{i=1}^{m} p_i \ln q_i \leq \sum_{i=1}^{m} p_i \ln p_i,$$

avec égalité si et seulement si pour tout i, l'égalité $p_i = q_i$ est satisfaite.

En tenant compte de la définition des q_i, et en multipliant les deux membres de cette inégalité par $-1/\ln q$, on obtient

$$\sum_{i=1}^{m} p_i n_i \geq -\sum_{i=1}^{m} p_i \frac{\ln p_i}{\ln q},$$

qui se réécrit immédiatement en

$$\overline{n}_{\mathbf{c}} \geq \frac{H(S)}{\log_2 q},$$

avec égalité si et seulement si l'égalité $p_i = q_i$, autrement dit $n_i = -\log_q p_i$ est satisfaite pour tout entier i compris entre 1 et m. Ce qui achève la démonstration du premier point du théorème.

3.3.4 Remarque sur le cas d'égalité

Une conséquence immédiate du théorème 3.1 est que, si la longueur moyenne d'un codage atteint la valeur minimale donnée par ce théorème, c'est-à-dire si

$$\frac{H(S)}{\log_2 q} = \overline{n}_{\mathbf{c}},$$

alors le codage est optimal.

En revanche la réciproque n'est pas vraie. Par exemple, pour la source définie par

$$\mathcal{S} = \{s_1, s_2\}$$

et

s	s_1	s_2
$\Pr(S = s)$	$\frac{3}{4}$	$\frac{1}{4}$

,

l'entropie est égale à

$$H(S) = -\left(\frac{3}{4}\log_2\frac{3}{4} + \frac{1}{4}\log_2\frac{1}{4}\right) = \log_2 4 - \frac{3}{4}\log_2 3 \approx 0,811,$$

et les codages binaires optimaux sont les codages de longueur fixe égale à 1, dont la longueur moyenne est évidemment strictement plus grande que $H(S)$.

3.3.5 Remarques sur la distribution de probabilité d'une source

Si un symbole s d'une source S a une probabilité nulle, la longueur du mot qui lui est associé dans un codage n'a aucune influence sur la longueur moyenne de ce codage. D'ailleurs, ce symbole n'étant jamais émis par la source, on peut toujours considérer qu'il n'est pas nécessaire de le coder.

Convention 1. *Désormais, toutes les sources considérées ne comprendront aucun symbole de probabilité nulle.*

Une conséquence de cette convention est qu'hormis pour le cas d'une source ne comprenant qu'un seul symbole, aucun symbole n'a une probabilité égale à 1.

Convention 2. *Toutes les sources considérées par la suite comprendront au moins deux symboles ($m \geq 2$).*

En conséquence, la quantité d'information contenue dans un symbole d'une source est finie et non nulle.

3.3.6 Seconde partie du théorème

Théorème 3.2. *Soit $S = (\mathcal{S}, \mathrm{P})$ une source.*

Il existe un codage $\mathbf{c}$ de S sur un alphabet cible à $q \geq 2$ éléments, tel que

$$\overline{n}_{\mathbf{c}} < \frac{H(S)}{\log_2 q} + 1.$$

Démonstration. Compte tenu des conventions adoptées pour les sources, pour i compris entre 1 et m, il existe un entier n_i tel que

$$-\log_q p_i \leq n_i < -\log_q p_i + 1.$$

On peut déduire de l'inégalité gauche de cet encadrement, que

$$\log_q p_i \geq -n_i \iff p_i \geq q^{-n_i}$$

Ce qui nous donne, en faisant la somme sur tous les symboles :

$$\sum_{i=1}^{m} q^{-n_i} \leq 1$$

et par conséquent, d'après le théorème de MacMillan (resp. Kraft), qu'il existe un code (resp. code préfixe) contenant des mots de longueur n_1, n_2, ..., n_m.

Si on code le symbole de probabilité p_i avec un mot de longueur n_i d'un tel code, on obtient un codage $\mathbf{c}$ de la source S dont la longueur moyenne est majorée par

$$\overline{n}_{\mathbf{c}} < \sum_{i=1}^{m} p_i(-\log_q p_i + 1) = \frac{H(S)}{\log_2 q} + 1.$$

Le théorème est ainsi démontré.

□

Remarque : lorsque l'alphabet cible est l'alphabet binaire ($q = 2$), le théorème s'énonce

— Il existe un codage *binaire* $\mathbf{c}$ de S, tel que

$$\overline{n}_{\mathbf{c}} < H(S) + 1.$$

3.3.7 Conséquence des théorèmes 3.1 et 3.2

Corollaire 3.1. *La longueur moyenne d'un codage optimal d'une source S satisfait l'encadrement*

$$\frac{H(S)}{\log_2 q} \leq \overline{n}_{\mathbf{c}} < \frac{H(S)}{\log_2 q} + 1.$$

3.4 Construction de codages optimaux

Une conséquence des théorèmes de Kraft et McMillan est qu'à tout codage optimal d'une source S correspond un codage préfixe avec des mots de même longueur, et par conséquent lui aussi optimal. Ce qui signifie qu'on peut limiter la recherche des codages optimaux à la classe des codages préfixes, sans rien perdre du point de vue de la longueur moyenne obtenue.

La remarque faite sur le cas d'égalité du théorème 3.1 apporte un moyen simple de créer un codage binaire optimal pour toute source S dont la distribution de probabilité est constituée de probabilités qui sont des puissances de 2. Pour une telle distribution, on obtient un codage optimal en associant à un symbole de probabilité $p = \frac{1}{2^n}$ un mot binaire de longueur $n = -\log_2 p$.

Pour les autres distributions, la construction doit s'obtenir différemment étant donné que $-\log_2 p$ n'est en général pas un nombre entier.

Nous désignerons dans la suite par n_1, n_2, ..., n_m les longueurs des mots associés aux symboles s_1, s_2, ..., s_m de la source S et nous supposerons que les probabilités de ces symboles sont dans l'ordre décroissant des indices

$$p_1 \geq p_2 \geq \ldots \geq p_m.$$

3.4.1 Quelques propriétés des codages optimaux

Lemme 3.2. *Dans un codage optimal* $\mathbf{c}$*, un symbole* s *de probabilité strictement plus petite que celle d'un symbole* s' *est codé par un mot au moins aussi long que le mot codant* s'*. Autrement dit, si* $p_i < p_j$ *alors* $n_i \geq n_j$*.*

Démonstration. Supposons que cela ne soit pas le cas, c'est-à-dire que l'on a deux indices i et j tels que $p_i < p_j$ et $n_i < n_j$.

On peut alors définir un autre codage $\mathbf{c}'$ par

$$\mathbf{c}'(s) = \mathbf{c}(s)$$

pour tout symbole $s \in \mathcal{S}$ autre que s_i et s_j, et par

$$\begin{aligned} \mathbf{c}'(s_i) &= \mathbf{c}(s_j) \\ \mathbf{c}'(s_j) &= \mathbf{c}(s_i). \end{aligned}$$

La différence des longueurs moyennes de ces deux langages est

$$\begin{aligned} \overline{n}_{\mathbf{c}'} - \overline{n}_{\mathbf{c}} &= p_i \times n_j + p_j \times n_i - p_i \times n_i - p_j \times n_j \\ &= (p_i - p_j)(n_j - n_i). \end{aligned}$$

On en déduit que

$$\overline{n}_{\mathbf{c}'} < \overline{n}_{\mathbf{c}},$$

ce qui contredit l'optimalité de $\mathbf{c}$. □

D'après ce lemme, on peut toujours supposer que les longueurs des mots d'un codage optimal sont dans l'ordre croissant des indices

$$n_1 \leq n_2 \leq \ldots \leq n_m.$$

Lemme 3.3. *Dans un codage préfixe optimal, les deux symboles les moins probables sont codés par des mots de même longueur. Autrement dit,*

$$n_{m-1} = n_m.$$

Démonstration. Compte tenu de la remarque qui suit le lemme 3.2, il suffit de prouver qu'il n'est pas possible d'avoir $n_{m-1} < n_m$ pour un codage préfixe optimal.

En effet, si on a $n_{m-1} < n_m$, alors on peut construire le codage $\mathbf{c}'$ défini par

$$\mathbf{c}'(s) = \mathbf{c}(s)$$

pour tout symbole s autre que s_m, et pour le symbole s_m par

$$\mathbf{c}'(s_m) = u$$

où u est le mot obtenu en retirant la dernière lettre de $\mathbf{c}(s_m)$.

Il est clair que $\mathbf{c}'$ est un codage préfixe dont la longueur moyenne est strictement plus petite que celle de $\mathbf{c}$. Ce qui achève la démonstration. □

Lemme 3.4. *Parmi les mots de longueur maximale d'un codage préfixe optimal, il en existe deux qui ne diffèrent que par la dernière lettre.*

Démonstration. Si ce n'était pas le cas, on obtiendrait un codage de longueur moyenne strictement plus petite en remplaçant tous les mots de longueur maximale par ces mots privés de leur dernière lettre. Ce qui contredirait l'optimalité du codage initial. □

3.4.2 Algorithme de Huffman

Remarque : dans toute cette partie, les codages considérés seront des codages binaires, $q = 2$.

Un algorithme dû à Huffman (1952) permet de construire un codage préfixe binaire optimal à partir de toute source.

Il s'appuie sur les deux lemmes qui suivent.

Lemme 3.5. *Les seuls codages binaires optimaux d'une source à deux symboles sont les deux codages de longueur fixe égale à 1.*

Lemme 3.6. *Si* $\mathbf{c}$ *est un codage préfixe binaire optimal pour une source* $S = (\mathcal{S} = \{s_1, \ldots, s_m\}, \mathrm{P} = \{p_1, \ldots, p_m\})$*, et si* p_i *peut être décomposée en une somme de deux probabilités non nulles* p *et* p' *inférieures ou égales à la probabilité* p_m *du symbole* s_m *le moins probable de* S*, alors on obtient un codage préfixe binaire optimal* $\mathbf{c}'$ *pour la source* S' *dont l'alphabet est obtenu en remplaçant* s_i *par deux nouveaux symboles* s *et* s'

$$\mathcal{S}' = \mathcal{S} \setminus \{s_i\} \cup \{s, s'\},$$

et la distribution de probabilité est

$$\mathrm{P}' = \{p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_m, p, p'\}$$

en posant

$$\mathbf{c}'(s_j) = \mathbf{c}(s_j),$$

pour tout $j \neq i$*, et*

$$\begin{aligned} \mathbf{c}'(s) &= \mathbf{c}(s_i).0 \\ \mathbf{c}'(s') &= \mathbf{c}(s_i).1. \end{aligned}$$

Démonstration. On suppose que le codage $\mathbf{c}'$ n'est pas optimal. Soit donc un codage préfixe binaire optimal $\mathbf{c'_1}$ de S'. Sa longueur moyenne est strictement plus petite que celle de $\mathbf{c}'$

$$\overline{n}_{\mathbf{c'_1}} < \overline{n}_{\mathbf{c}'}.$$

D'après les lemmes 3.2, 3.3 et 3.4, on peut supposer que les deux mots $\mathbf{c'_1}(s) = u.0$ et $\mathbf{c'_1}(s') = u.1$ ne diffèrent que par leur dernier bit.

Considérons le codage $\mathbf{c_1}$ de la source initial S défini par

$$\mathbf{c_1}(s_j) = \mathbf{c'_1}(s_j)$$

pour $j \neq i$, et

$$\mathbf{c_1}(s_i) = u.$$

Le calcul de la longueur moyenne de $\mathbf{c_1}$ donne

$$\begin{aligned} \overline{n}_{\mathbf{c_1}} &= \overline{n}_{\mathbf{c'_1}} - p|u.0| - p'|u.1| + p|u| + p'|u| \\ &= \overline{n}_{\mathbf{c'_1}} - (p + p') \\ &= \overline{n}_{\mathbf{c'_1}} - p_i \\ &< \overline{n}_{\mathbf{c}'} - p_i \\ &= \overline{n}_{\mathbf{c}}. \end{aligned}$$

Ainsi, le codage $\mathbf{c_1}$ obtenu pour la source S a une longueur moyenne strictement plus petite que celle de $\mathbf{c}$. Cela contredit l'hypothèse que $\mathbf{c}$ est optimal, et notre hypothèse initiale doit être rejetée. Le codage $\mathbf{c}'$ est optimal. □

Une conséquence de ces deux lemmes est qu'un codage préfixe binaire optimal peut être construit de proche en proche par un algorithme appelé *algorithme de Huffman*, et le codage produit par cet algorithme est appelé *codage de Huffman*.

Soit la source $S = (\mathcal{S}, \mathrm{P})$

s	s_1	s_2	s_3	s_4	s_5	s_6
$\Pr(S = s)$	$0,3$	$0,2$	$0,15$	$0,15$	$0,1$	$0,1$

L'entropie de cette source vaut

$$H(S) \approx 2,471.$$

La construction d'un codage de Huffman pour cette source est :

symboles	s_1	s_2	s_3	s_4	s_5	s_6
proba	$0,3$	$0,2$	$0,15$	$0,15$	$0,1$	$0,1$
lettres					0	1
symboles	s_1	s_2	s_{56}	s_3	s_4	
proba	$0,3$	$0,2$	$0,2$	$0,15$	$0,15$	
lettres				0	1	
symboles	s_1	s_{34}	s_2	s_{56}		
proba	$0,3$	$0,3$	$0,2$	$0,2$		
lettres			0	1		
symboles	s_{256}	s_1	s_{34}			
proba	$0,4$	$0,3$	$0,3$			
lettres		0	1			
symboles	s_{134}	s_{256}				
proba	$0,6$	$0,4$				
lettres	0	1				

Il suffit ensuite de reconstruire les mots de code :

symboles	s_{134}	s_{256}				
codages	0	1				
symboles	s_{256}	s_1	s_{34}			
codages	1	00	01			
symboles	s_1	s_{34}	s_2	s_{56}		
codages	00	01	10	11		
symboles	s_1	s_2	s_{56}	s_3	s_4	
codages	00	10	11	010	011	
symboles	s_1	s_2	s_3	s_4	s_5	s_6
codages	00	10	010	011	110	111

La longueur moyenne de ce codage est

$$\overline{n}_{\mathbf{c}} = 2,5.$$

Une autre façon de présenter la construction d'un codage de Huffman consiste à associer à chaque symbole des sources successives un arbre binaire.

Pour la première source, on associe à chaque symbole un arbre réduit à un seul nÖud. Ainsi pour notre exemple, la « forêt » initiale est constituée de six arbres (voir figure 3.1).

On réunit les deux arbres de probabilité les plus petites en un unique arbre, sa probabilité étant égale à la somme des deux probabilités des arbres réunis. Sur notre exemple, la forêt ne comprend alors plus que cinq arbres (voir figure 3.2).

FIGURE 3.1 – Forêt initiale

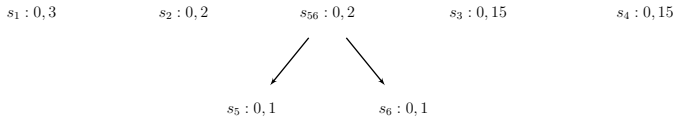


FIGURE 3.2 – Forêt obtenue après la première étape

On procède ensuite de la même façon jusqu'à obtenir une forêt ne contenant plus qu'un seul arbre : c'est un arbre du codage de Huffman (voir figures 3.3, 3.4, 3.5, 3.6).

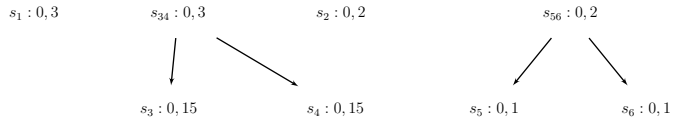


FIGURE 3.3 – Forêt obtenue après la deuxième étape

3.4.3 Application

Le codage de Huffman est couramment utilisé en compression des données. Il peut être utilisé seul ou en complément d'autres méthodes.

L'algorithme 3.1 page suivante donne dans les grandes lignes une méthode de compression d'un fichier vu comme une suite de N octets (l'alphabet source est donc l'ensemble des 256 octets).

La taille en octets du fichier compressé sera approximativement

$$T = \frac{N \times \overline{n}_{\mathbf{c}}}{8} + K.$$

La grandeur K est la taille en octets de stockage du codage de Huffman obtenu à la ligne 2 de l'algorithme 3.1 page suivante.

À titre d'exemple, un fichier contenant un texte français, codé en ISO-8859-1, a une entropie de 4,43 bits/octet approximativement. Par conséquent, la longueur moyenne d'un codage de Huffman des octets de ce fichier est comprise entre 4,43 et 5,43 bits/octet. Cela donne un taux de compression compris entre $1 - \frac{4,43}{8} = 44,6\%$ et $1 - \frac{5,43}{8} = 32,1\%$.

Si au lieu de considérer un tel fichier comme une suite d'octets mais comme une suite de couple d'octets (la source est alors constitué des $2^{16} = 65536$ couples d'octets), l'entropie d'un texte français codé en ISO-8859-1 est de 7,77 bits par couple d'octets. Le taux de compression obtenu par un codage de Huffman est alors compris entre $1 - \frac{7,77}{16} = 51,4\%$ et $\frac{8,77}{16} = 45,2\%$.

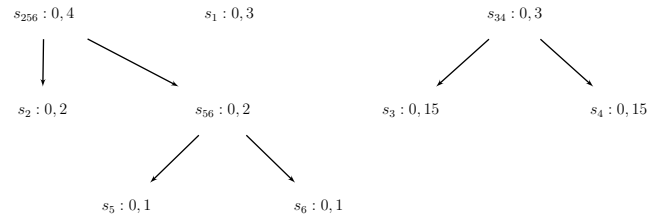


FIGURE 3.4 – Forêt obtenue après la troisième étape

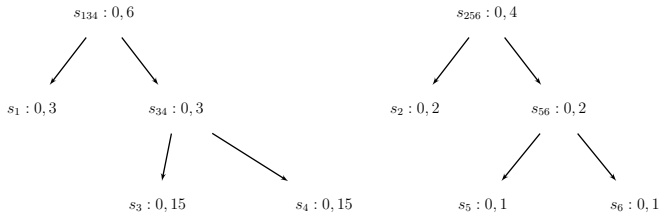


FIGURE 3.5 – Forêt obtenue après la quatrième étape

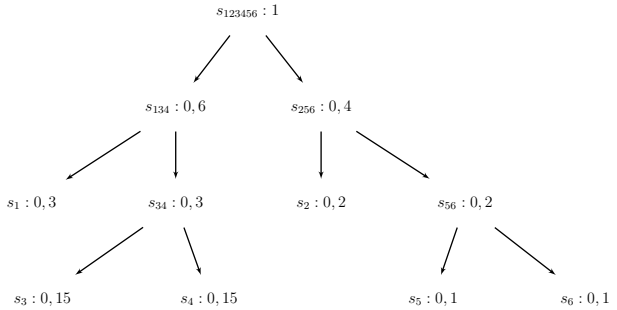


FIGURE 3.6 – Arbre du codage de Huffman obtenu

Algorithme 3.1 Algorithme de compression d'un fichier à l'aide du codage de Huffman

1: Parcourir l'intégralité du fichier pour compter le nombre d'occurrences de chaque octet.
2: Construire un codage de Huffman pour l'ensemble des octets à partir des fréquences observées.
3: Coder chacun des octets à l'aide du codage de Huffman obtenu.

3.5 Exercices

Exercice 3-1 *Pour s'échauffer*

Question 1 Quelle est la longueur moyenne d'un codage de longueur fixe ?

Question 2 Que dire de la longueur moyenne dans le cas de l'équiprobabilité ?

Exercice 3-2

Question 1

Montrez que dans un codage binaire optimal, l'inégalité de Kraft devient une égalité.

Question 2 Dans un codage binaire optimal peut–il y avoir trois mots de longueur maximale ?

Exercice 3-3

Soit $S = (\mathcal{S}, P)$ une source d'information. Montrez qu'il existe un codage préfixe binaire de cette source dans lequel tout symbole $s \in \mathcal{S}$ est codé par un mot de longueur $\lceil I(s) \rceil$.

Exercice 3-4

La distribution des longueurs des mots d'un codage préfixe binaire optimal est-elle unique ?

Exercice 3-5

Quels sont les arbres des codages préfixes binaires optimaux d'une source de trois symboles ? de quatre symboles ?

Exercice 3-6

Dans le cas où tous les caractères de l'alphabet S à coder sont équiprobables, quels sont les codages optimaux ?

Exercice 3-7 *Entropie d'un message*

On considère un message M écrit avec les symboles d'un alphabet $\mathcal{S} = \{s_1, s_2, \ldots, s_m\}$. On désigne par n_i le nombre d'occurrences du symbole s_i dans le message M. La longueur du message est désignée par N. On a évidemment

$$N = \sum_{i=1}^{m} n_i.$$

Exprimez l'entropie de M en fonction des entiers N et n_i, $i = 1, \ldots, m$.

Exercice 3-8

On considère une source de 8 symboles décrites par le tableau ci-dessous.

s	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8
$\Pr(S = s)$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$	$\frac{1}{16}$

Question 1 Déterminez un codage optimal pour cette source.

Question 2 Combien y a-t-il de codages binaires préfixes optimaux pour cette source ?

pour approfondir **Question 3** Généralisation de la question précédente pour des distributions dont les fréquences sont des puissances de 2.

Exercice 3-9 *Codage de Huffman*

On considère la source d'information définie par l'alphabet $\mathcal{S} = \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\}$ avec la distribution de probabilités suivante :

s	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8
$\Pr(S = s)$	0,4	0,18	0,1	0,1	0,07	0,06	0,05	0,04

Question 1 Calculez la quantité d'information contenue dans chacun des huit symboles, puis l'entropie de la source d'information $(\mathcal{S}, f)$.

Question 2 Calculez un codage de Huffman de cette source, puis la longueur moyenne de ce codage. Comparez cette longueur moyenne à l'entropie.

pour approfondir **Exercice 3-10**

Question 1 Même exercice pour la source (les données ne sont pas des fréquences mais des effectifs)

x	a	b	c	d	e	f	g	h
$f(x)$	1	1	2	3	5	8	13	21

Question 2 Les effectifs donnés dans le tableau ci–dessous sont les premiers termes de la suite de Fibonacci. Peut–on généraliser le codage pour un alphabet source de n symboles dont les fréquences sont les n premiers termes de la suite de Fibonacci ?

Exercice 3-11

Soit $\mathcal{S}$ la source munie de la distribution de fréquences f suivante :

x_1	x_2	x_3	x_4	x_5	x_6	x_7
1/8	1/32	1/4	1/8	1/16	5/32	1/4

Question 1 Peut-on trouver un codage binaire optimal de $\mathcal{S}$ comprenant

1. trois mots de longueur 2, deux mots de longueur 3 et deux mots de longueur 4 ?
2. un mot de longueur 2, cinq mots de longueur 3 et un mot de longueur 4 ?

Question 2 Construisez un codage binaire optimal pour $\mathcal{S}$.

Soient c_1 et c_2 les codages de $\mathcal{S}$ définis par

	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$c_1(x)$	0011	0000	10	01	0010	0001	11
$c_2(x)$	001	000000	1	0001	00001	000001	01

Question 3 Combien de bits faut–il en moyenne pour coder une séquence de 16 000 symboles à l'aide de ces deux codages ?

Question 4 Ces codages sont–ils optimaux ?

pour approfondir **Exercice 3-12** *Tailles d'un fichier*

L'analyse des caractères d'un fichier de programmes en PASCAL donne la répartition suivante (par ordre croissant des effectifs) :

W	$	Y	F	5	à	X	>	*	"
1	1	1	1	2	2	2	2	2	2
q	3	z	+	4	O	P	y	S	V
2	3	3	3	4	5	6	6	8	8
M	j	R	2	-	.	<	U	D	B
8	10	10	11	11	13	15	16	17	17
é	C	v	x	{	L	}	[	]	I
18	20	21	23	25	25	25	27	27	34
E	N	A	k	h	0	1	T	g	w
35	35	36	36	37	40	40	42	43	46
m	’	b	u	P	=	_	l	,	d
58	58	66	70	74	78	79	89	90	93
:	s	f	(	)	c	a	o	n	;
96	107	123	128	128	131	157	158	199	200
r	t	i	e	/	<ESP>				
216	273	278	283	304	1972				

<ESP> désigne le caractère espace.

Question 1 En supposant que chaque caractère est codé en code ASCII sur 8 bits, quelle est la taille de ce fichier (en bits) ?

Question 2 Quelle est la taille de ce fichier si chaque caractère est codé par un code de longueur fixe minimale ?

Question 3 Donner une minoration de la taille de ce fichier s’il est codé par un codage préfixe.

Question 4 Quelle est la taille de ce fichier s’il est codé par un codage de Huffman ?

Exercice 3-13

Soit un fichier de données contenant une séquence de caractères codés sur 8 bits telle que les 256 caractères apparaissent à peu près aussi souvent : la fréquence maximale d’un caractère est moins du double de la fréquence minimale. Prouvez que dans ce cas, le codage de Huffman n’est pas plus efficace que le codage de longueur fixe sur 8 bits.

Exercice 3-14 *Stratégie*

On a lancé deux fois une pièce de monnaie.

Question 1 Quelle est la quantité d’information contenue dans les affirmations suivantes :

1. PILE n’est pas apparu ?
2. PILE est apparu une seule fois ?
3. PILE est apparu deux fois ?

Question 2 Quelle est l’entropie de la source d’information indiquant le nombre de PILE ?

Question 3 Vous désirez connaître le nombre de PILE sortis lors de ces deux tirages. Pour cela vous pouvez poser autant de questions à réponses oui ou non que vous voulez. Quelle est la meilleure stratégie vous permettant de connaître ce nombre ? Par meilleure stratégie, on entend celle qui permet d’obtenir le nombre en un minimum de questions en moyenne.

Question 4 Reprendre le même exercice avec 3 lancers.

Chapitre 4

Détection et correction des erreurs

4.1 Nécessité de la détection/correction d'erreurs

Toute transmission d'information est susceptible d'être victime d'erreurs. Il est évidemment utile, voire indispensable pour certaines applications, de savoir qu'une information a mal été transmise. Il s'agit dans ce cas de détection d'erreur. Si on sait seulement que l'information est erronée il faut alors la transmettre à nouveau, en espérant qu'elle se fasse correctement. Une autre approche consiste à éviter un nouveau transfert des données, en les corrigeant directement après transfert.

Tout cela ne se fait pas « gratuitement » et nécessite de transmettre des informations supplémentaires qui permettront ensuite la détection ou la correction d'erreurs. C'est ce que nous allons développer dans ce chapitre.

4.2 Canal binaire symétrique sans mémoire

4.2.1 Canal de communication

On appelle *canal de communication* tout procédé permettant de transmettre d'un lieu à un autre (communication dans l'espace) ou d'un instant à un autre (communication dans le temps) les symboles s d'un alphabet $\mathcal{S}$. Ce canal peut être *bruité*. Ceci signifie que lors de la transmission un symbole s peut être transformé en un autre symbole. On parle alors d'*erreur* de transmission.

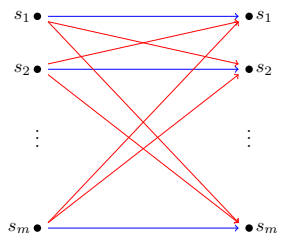


FIGURE 4.1 – Canal de communication

Un canal de communication est caractérisé par la donnée des probabilités d'erreurs de transmission pour chacun de ses symboles. On note $\Pr(s' \text{ reçu} | s \text{ envoyé})$ la probabilité que le symbole

s' soit reçu si le symbole s est émis dans le canal.

Exemple 1 :

Le canal représenté par la figure 4.2 permet la communication de trois symboles : $\mathcal{S} = \{0, 1, 2\}$.

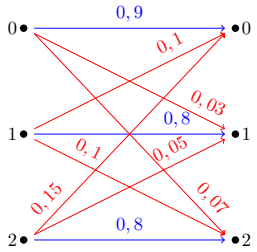


FIGURE 4.2 – Un exemple de canal bruité

Dans ce canal les probabilités de transmissions sont

$$\begin{array}{lll} \Pr(0 \text{ reçu}|0 \text{ envoyé}) = 0,9 & \Pr(0 \text{ reçu}|1 \text{ envoyé}) = 0,1 & \Pr(0 \text{ reçu}|2 \text{ envoyé}) = 0,15 \\ \Pr(1 \text{ reçu}|0 \text{ envoyé}) = 0,03 & \Pr(1 \text{ reçu}|1 \text{ envoyé}) = 0,8 & \Pr(1 \text{ reçu}|2 \text{ envoyé}) = 0,05 \\ \Pr(2 \text{ reçu}|0 \text{ envoyé}) = 0,07 & \Pr(2 \text{ reçu}|1 \text{ envoyé}) = 0,1 & \Pr(2 \text{ reçu}|2 \text{ envoyé}) = 0,8. \end{array}$$

4.2.2 Canal binaire

Le canal est dit *binaire* si les symboles qu'il permet de transmettre sont des bits ($\mathcal{S} = \{0, 1\}$). Un canal binaire est entièrement caractérisé par la probabilité p_0 qu'un bit 0 transmis soit reçu comme un bit 1

$$p_0 = \Pr(1|0),$$

et la probabilité p_1 qu'un bit 1 transmis soit reçu comme un bit 0

$$p_1 = \Pr(0|1).$$

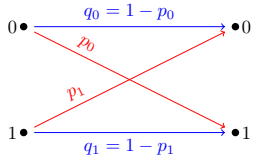


FIGURE 4.3 – Canal binaire

4.2.3 Canal binaire symétrique

Un canal binaire *symétrique* est un canal binaire dans lequel la probabilité p d'erreur dans la transmission d'un bit est la même pour chacun des deux bits

$$p_0 = p_1 = p.$$

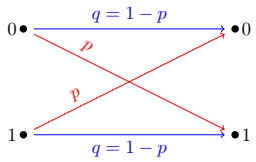


FIGURE 4.4 – Canal binaire symétrique

Un canal binaire symétrique est donc entièrement caractérisé par la probabilité p d'erreur lors de la transmission d'un bit.

On note $q = 1 - p$ la probabilité de transmission correcte d'un bit.

4.2.4 Canal sans mémoire

Un canal est dit *sans mémoire* si les erreurs éventuelles de transmission de bits sont indépendantes les unes des autres. Autrement dit, le canal est sans mémoire, si pour tout couple de mots de longueur n, $\mathbf{v} = v_1, v_2, \ldots, v_n$ et $\mathbf{v}' = v'_1, v'_2, \ldots, v'_n$, on a

$$\Pr(\mathbf{v}' \text{ reçu}|\mathbf{v} \text{ envoyé}) = \prod_{i=1}^{n} \Pr(v'_i \text{ reçu}|v_i \text{ envoyé}).$$

Exemple 2 :
En prenant le canal décrit dans l'exemple 1, si ce canal est sans mémoire, alors la probabilité de recevoir le mot $\mathbf{v}' = 021$, alors que le mot $\mathbf{v} = 012$ est envoyé est

$$\Pr(\mathbf{v}' \text{ reçu}|\mathbf{v} \text{ envoyé}) = 0,9 \times 0,1 \times 0,05 = 0,0045.$$

Pour désigner les canaux binaires symétriques sans mémoire nous utiliserons l'acronyme CBSSM.

Théorème 4.1. *Dans un CBSSM de probabilité d'erreur p, la probabilité qu'un mot binaire de longueur n, $\mathbf{v} = v_1, v_2, \ldots, v_n$, soit reçu en un mot binaire $\mathbf{v}' = v'_1, v'_2, \ldots, v'_n$ est égale à*

$$\Pr(\mathbf{v}' \textit{ reçu}|\mathbf{v} \textit{ envoyé}) = p^d q^{n-d},$$

où d est le nombre d'erreurs, c'est-à-dire le nombre d'indices i pour lesquels $u_i \neq u'_i$.

Démonstration. C'est une conséquence immédiate de la propriété d'indépendance des erreurs de transmission de bits. Car pour chacun des couples de bits, on a

$$\Pr(u'_i \text{ reçu}|u_i \text{ envoyé}) = q$$

si $u_i = u'_i$ (transmission sans erreur), et on a

$$\Pr(u'_i \text{ reçu}|u_i \text{ envoyé}) = p$$

si $u'_i \neq u_i$ (transmission erronée). □

Corollaire 4.1. *En particulier, la probabilité qu'un mot $\mathbf{v}$ envoyé soit correctement reçu est*

$$\Pr(\mathbf{v} \textit{ reçu}|\mathbf{v} \textit{ envoyé}) = q^n.$$

Et comme il est bien connu que

$$\lim_{n \to +\infty} q^n = 0$$

lorsque $0 \leq q < 1$, hormis dans le cas d'un CBSSM de probabilité d'erreur $p = 0$, il est virtuellement certain que des erreurs se produisent lorsqu'un très grand nombre de bits sont transmis.

4.3 Codage par blocs

Dorénavant, nous noterons $\mathbb{F}_2$ l'alphabet binaire. Ainsi, lorsque n est un entier naturel, $\mathbb{F}_2^n$ désigne l'ensemble des mots binaires de longueur n.

4.3.1 Principe du codage par bloc

De nombreux codages détecteurs/correcteurs sont conçus comme des *codages par blocs*. L'idée consiste à coder non pas les bits individuellement, mais à les coder par blocs d'une certaine longueur fixée. L'information est ainsi découpée en blocs tous de même longueur, notons la k, et on code séparément chacun des blocs ainsi obtenus.

Afin de pouvoir détecter, voire corriger des erreurs de transmission, les codages augmentent toujours la longueur des mots en ajoutant des bits qui introduisent de la redondance, et permettent, lorsqu'ils sont savamment choisis, de contrôler l'exactitude ou non des mots reçus [1].

L'augmentation de taille des mots lors de l'opération de codage est le plus souvent toujours la même, et donc les blocs sont tous codés par un mot de la même longueur n. En notant r cette augmentation de taille, on a la relation

$$n = k + r.$$

Les codages par blocs sont donc des codages de longueur fixes, et par conséquent peuvent a priori être n'importe quelle application injective de l'ensemble des mots binaires de longueur k dans celui des mots de longueur n.

$$\mathbf{c} \ : \ \mathbb{F}_2^k \longrightarrow \mathbb{F}_2^n.$$

Le *rendement* (ou encore *taux de transmission*) d'un codage est le rapport de la longueur des mots à coder à celle des mots du code

$$R = \frac{k}{n}.$$

Il exprime en quelque sorte la quantité d'information par bit transmis.

Il est bien entendu souhaitable que le rendement d'un code soit le plus élevé possible. Mais pour une longueur fixée des mots du code, cet objectif est contradictoire avec celui de la capacité à détecter/corriger les erreurs.

4.3.2 Codage systématique

Un codage par bloc est dit *systématique* si pour tout mot $\mathbf{u} \in \mathbb{F}_2^k$, le mot que lui associe le codage est de la forme

$$\mathbf{c}(\mathbf{u}) = \mathbf{u}.\mathbf{r},$$

autrement dit si $\mathbf{u}$ est un préfixe du mot $\mathbf{v}$ qui le code, le mot $\mathbf{r}$ qui suit pouvant (a priori) être n'importe quel mot de longueur $r = n - k$.

1. Les codages optimaux ont pour but de supprimer la redondance existante dans les sources d'information. Les codages détecteurs/correcteurs d'erreurs font le contraire.

4.3.3 Poids et distance de Hamming

La *distance de Hamming* entre deux mots $\mathbf{v} = v_1 v_2 \dots v_n$ et $\mathbf{v}' = v'_1 v'_2 \dots v'_n$ de $\mathbb{F}_2^n$ est le nombre d'indices i pour lesquels $v_i \neq v'_i$. Cette distance est notée $\mathrm{d}_H(\mathbf{v}, \mathbf{v}')$.

$$\mathrm{d}_H(\mathbf{v}, \mathbf{v}') = \mathrm{card}(\{i \in [\![1, n]\!] \mid v_i \neq v'_i\}).$$

Le *poids de Hamming* d'un mot $\mathbf{v}$ de $\mathbb{F}_2^n$ est le nombre d'indices i pour lesquels $v_i \neq 0$. Le poids de Hamming d'un mot est noté $\omega_H(\mathbf{v})$.

$$\omega_H(\mathbf{v}) = \mathrm{card}(\{i \in [\![1, n]\!] \mid v_i \neq 0\}).$$

Proposition 4.1. *Soient* $\mathbf{v}$ *et* $\mathbf{v}'$ *deux mots de* $\mathbb{F}_2^n$.

$$\mathrm{d}_H(\mathbf{v}, \mathbf{v}') = \omega_H(\mathbf{v} \oplus \mathbf{v}').$$

4.3.4 Mot d'erreur

Si à la sortie d'un CBSSM on reçoit un mot $\mathbf{v}'$ lorsqu'un mot $\mathbf{v}$ a été envoyé, la différence entre ces deux mots peut s'écrire sous la forme d'un ou-exclusif bit à bit entre ces deux mots. Cette différence est un mot $\mathbf{e}$ de même longueur que $\mathbf{v}$ et $\mathbf{v}'$ que l'on nomme *mot d'erreur*. On a la relation

$$\mathbf{e} = \mathbf{v} \oplus \mathbf{v}',$$

ou encore

$$\mathbf{v}' = \mathbf{v} \oplus \mathbf{e}.$$

Dans le théorème 4.1, l'entier d dans la probabilité de recevoir le mot $\mathbf{v}'$ en sachant que le mot $\mathbf{v}$ a été envoyé est le poids de l'erreur $\mathbf{e}$.

Nous terminons cette section par deux exemples classiques de codage détecteur/correcteur d'erreurs.

4.3.5 Bit de parité

La famille des codages de parité est constituée de codages systématiques dans lesquels le mot $\mathbf{r}$ ajouté à la fin de chaque mot $\mathbf{u} \in \mathbb{F}_2^k$ à coder est un mot d'un seul bit choisi de sorte que le mot ainsi obtenu soit de poids pair. Les mots du codage sont ainsi de longueur $n = k + 1$, et le rendement du codage est

$$R = \frac{k}{k+1}.$$

Exemple 3 :
Avec $k = 3$, on obtient le codage suivant.

$\mathbf{u}$	$\mathbf{v} = \mathbf{c}(\mathbf{u})$
000	0000
001	0011
010	0101
011	0110
100	1001
101	1010
110	1100
111	1111

Le rendement du codage est $R = \frac{3}{4} = 75\%$.

Tous les mots du code associé sont donc de poids pair. Inversement tout mot $\mathbf{v}$ de poids pair est un mot du code, c'est le mot qui code le mot $\mathbf{u}$ obtenu en supprimant le dernier bit de $\mathbf{v}$. Ainsi le code est l'ensemble des mots de poids pair et de longueur $n = k + 1$. Nous noterons Parite(k) ce code. Par exemple, Parite(3) est constitué de l'ensemble des 8 mots de longueur 4 et de poids pair.

Lorsqu'un mot $\mathbf{v}' \in \mathbb{F}_2^n$ est reçu, en vérifiant la parité de son poids, on sait si ce mot est ou n'est pas dans le code :

— Si $\omega_H(\mathbf{v}')$ est impair, alors $v' \notin$ Parite(k), et il est alors certain que le mot envoyé a subi des erreurs. Dit autrement, la probabilité que le mot $\mathbf{v}'$ soit correct, c.-à-d. égal au mot envoyé, est nulle.

$$\Pr(\mathbf{v}' \text{ correct}) = 0.$$

Le mot $\mathbf{v}'$ est ainsi reconnu comme erroné.

— Si au contraire $\omega_H(\mathbf{v}')$ est pair, alors il est possible que ce mot soit correct, mais il est possible aussi qu'il ait été bruité par un mot d'erreur de poids pair. Rien ne peut être affirmé avec certitude.

4.3.6 Répétition

Les codages par répétition forment une autre famille de codages systématiques dans lesquels le mot à coder $\mathbf{u} \in \mathbb{F}_2^k$ est répété p fois, p étant un nombre entier qui détermine le codage.

$$\mathbf{c}(\mathbf{u}) = \mathbf{u}.\mathbf{u}.\ldots.\mathbf{u} = \mathbf{u}^p.$$

(Le mot $\mathbf{r}$ ajouté après $\mathbf{u}$ est donc $\mathbf{r} = \mathbf{u}^{p-1}$)

La longueur des mots du codage est donc $n = p \cdot k$, et le code est constitué de l'ensemble de tous les mots de la forme $\mathbf{u}^p$. Nous noterons Rep(k, p) ce code. Il contient 2^k mots de longueur $n = p \cdot k$.

Le rendement de ces codages est

$$R = \frac{1}{p}.$$

Exemple 4 :
Avec $k = 1$ et $p = 3$, on obtient le codage suivant

$\mathbf{u}$	$\mathbf{v} = \mathbf{c}(\mathbf{u})$
0	000
1	111

dont le rendement n'est que de $R = \frac{1}{3} \approx 33\%$.

Le code ici ne contient que les deux mots

$$\text{Rep}(1, 3) = \{000, 111\}.$$

4.4 Détection des erreurs

Détecter une erreur dans un mot reçu $\mathbf{v}'$, c'est constater que ce mot n'est pas dans le code.

Exemple 5 :

1. Avec le code de parité $C = \text{Parite}(k)$, si un mot $\mathbf{v} \in C$ est envoyé et qu'un mot d'erreur $\mathbf{e}$ de poids impair produit le mot $\mathbf{v}' = \mathbf{v} \oplus \mathbf{e}$, comme $\mathbf{v}' \notin C$ (puisque de poids impair), l'erreur est détectée.
2. Avec le code de répétition $C = \text{Rep}(k, p)$, si un mot $\mathbf{v} \in C$ est envoyé et qu'un mot d'erreur $\mathbf{e}$ de poids inférieur ou égal à $p-1$ produit le mot $\mathbf{v}' = \mathbf{v} \oplus \mathbf{e}$, comme $\mathbf{v}' \notin C$ (puisque pas de la forme $\mathbf{u}^p$), l'erreur est détectée.

Un code est *au moins t-détecteur* si toute erreur de poids au plus t est détectée. Autrement dit si

$$\forall \mathbf{v} \in C,\ \forall \mathbf{e} \in \mathbb{F}_2^n,\ 1 \leq \omega_H(\mathbf{e}) \leq t \Rightarrow \mathbf{v}' = \mathbf{v} \oplus \mathbf{e} \notin C.$$

Un code est *exactement t-détecteur* s'il est au moins t-détecteur mais pas $t+1$-détecteur.

Exemple 6 :

1. Parite(k) est un code au moins 1-détecteur, car toute erreur de poids égal à 1 survenant sur un mot de poids pair produit un mot de poids impair qui n'est pas dans le code. En revanche, les erreurs de poids 2 produisent des mots de poids pair, et ces erreurs ne sont donc pas détectées. Parite(k) est donc un code exactement 1-détecteur.
2. Rep(k, p) est un code au moins $p-1$-détecteur. Mais il n'est pas au moins p-détecteur, car si $\mathbf{v} \in \text{Rep}(k, p)$ et si $\mathbf{e} = \mathbf{u}^p$ avec $\omega_H(\mathbf{u}) = 1$, alors $\mathbf{v}' = \mathbf{v} \oplus \mathbf{e} \in \text{Rep}(k, p)$. Donc Rep$(k, p)$ est exactement $p-1$-détecteur.
3. Le code $C = \{\texttt{0000}, \texttt{0111}, \texttt{1010}, \texttt{1111}\}$ n'est pas au 1-détecteur (car `0000` et `0111` sont à une distance 1).
4. Le code $C = \{\texttt{00000}, \texttt{01101}, \texttt{10110}, \texttt{11011}\}$ est exactement 2-détecteur.

4.5 Correction des erreurs

Le problème de la correction est plus complexe que celui de la détection. Il s'agit, connaissant le mot $\mathbf{v}'$ reçu, et pour lequel on a détecté une erreur, de retrouver le mot envoyé $\mathbf{v}$ et par conséquent l'erreur $\mathbf{e}$. Ce problème consiste donc en une seule équation

$$\mathbf{v}' = \mathbf{v} \oplus \mathbf{e},$$

dans laquelle il y a deux inconnues : $\mathbf{v}$ et $\mathbf{e}$. Il est clair qu'on ne peut pas le résoudre au sens classique, puisque pour tout mot $\mathbf{v}$ du code, il existe toujours un mot $\mathbf{e}$ de longueur n pour lequel l'équation précédente est satisfaite. Il suffit de prendre

$$\mathbf{e} = \mathbf{v} \oplus \mathbf{v}'.$$

Alors que faire ? Comment corriger un mot erroné ? Parmi tous les mots $\mathbf{v}$ du code, lequel choisir ? Peut-on toujours le faire ?

Pour répondre à ces questions nous allons donner deux réponses équivalentes sous certaines hypothèses.

4.5.1 Correction par maximum de vraisemblance

Corriger un mot erroné $\mathbf{v}'$, c'est choisir parmi tous les mots $\mathbf{v}$ du code celui qui est le plus probable en sachant que l'on a reçu $\mathbf{v}'$. Autrement dit, on cherche le mot $\mathbf{v}$ pour lequel on a

$$\Pr(\mathbf{v} \text{ envoyé}|\mathbf{v}' \text{ reçu}) = \max\{\Pr(\mathbf{w} \text{ envoyé}|\mathbf{v}' \text{ reçu}) \mid \mathbf{w} \in C\}.$$

Exemple 7 :
Prenons le codage par répétition 3 fois d'un bit. Le code est

$$C = \mathrm{Rep}(1,3) = \{\mathtt{000}, \mathtt{111}\}.$$

Supposons que nous ayons reçu le mot $\mathbf{v}' = \mathtt{010}$. Calculons les probabilités respectives que l'un des deux mots aient été envoyés en sachant qu'on a reçu le mot $\mathbf{v}'$.

— Pour le mot $\mathtt{000}$, on a

$$\Pr(\mathtt{000} \text{ env}|\mathbf{v}' \text{ reçu}) = \frac{\Pr(\mathbf{v}' \text{ reçu}|\mathtt{000} \text{ env}) \cdot \Pr(\mathtt{000} \text{ env})}{\Pr(\mathbf{v}' \text{ reçu}|\mathtt{000} \text{ env}) \cdot \Pr(\mathtt{000} \text{ env}) + \Pr(\mathbf{v}' \text{ reçu}|\mathtt{111} \text{ env}) \cdot \Pr(\mathtt{111} \text{ env})},$$

ce qui donne après simplification en supposant l'équiprobabilité d'envoi des deux mots du code

$$\Pr(\mathtt{000} \text{ env}|\mathbf{v}' \text{ reçu}) = \frac{\Pr(\mathbf{v}' \text{ reçu}|\mathtt{000} \text{ env})}{\Pr(\mathbf{v}' \text{ reçu}|\mathtt{000} \text{ env}) + \Pr(\mathbf{v}' \text{ reçu}|\mathtt{111} \text{ env})}.$$

On a bien entendu

$$\Pr(\mathbf{v}' \text{ reçu}|\mathtt{000} \text{ env}) = \Pr(\mathbf{e} = \mathtt{010}) = p(1-p)^2,$$

et

$$\Pr(\mathbf{v}' \text{ reçu}|\mathtt{111} \text{ env}) = \Pr(\mathbf{e} = \mathtt{101}) = p^2(1-p).$$

Ce qui donne finalement

$$\Pr(\mathbf{v}' \text{ reçu}|\mathtt{000} \text{ env}) = 1 - p.$$

— Pour le mot $\mathtt{111}$, par un calcul analogue, on a

$$\Pr(\mathtt{000} \text{ env}|\mathbf{v}' \text{ reçu}) = p.$$

La plus petite des deux probabilités est la deuxième si on fait l'hypothèse raisonnable que $p < \frac{1}{2}$. Selon le principe de correction par lmaximum de vraisemblance, on corrige le mot $\mathbf{v}'$ en le remplaçant par $\mathtt{000}$.

4.5.2 Correction par proximité

Corriger un mot erroné $\mathbf{v}'$, c'est choisir parmi tous les mots $\mathbf{v}$ du code celui qui est le plus proche selon la distance de Hamming.

Exemple 8 :

1. Dans l'exemple 7, la correction du mot $\mathbf{v}' = \mathtt{010}$ donne le mot $\mathtt{000}$. On peut remarquer que ce dernier mot est de tous les mots du code celui qui est le plus proche de $\mathbf{v}'$.

Théorème 4.2. *Pour un CBSSM de probabilité* $p < \frac{1}{2}$*, et sous l'hypothèse que les* 2^k *mots de* $\mathbb{F}_2^k$ *sont équiprobables, alors la correction par maximum de vraisemblance est équivalente à la correction par proximité. Autrement dit, le mot* $\mathbf{v}$ *du code le plus proche d'un mot* $\mathbf{v}'$ *est aussi celui qui maximise la probabilité* $\Pr(\mathbf{v}$ *envoyé*$|\mathbf{v}'$ *reçu*$)$.

4.5.3 Boules

Étant donnés un mot $\mathbf{v} \in \mathbb{F}_2^n$ et un réel $r \geq 0$, on appelle *boule* de centre $\mathbf{v}$ et de rayon r, l'ensemble de tous les mots de $\mathbb{F}_2^n$ situés à une distance de Hamming de $\mathbf{v}$ inférieure ou égale à r. En notant les boules $\mathcal{B}(\mathbf{v}, r)$, on a

$$\mathcal{B}(\mathbf{v}, r) = \{\mathbf{w} \in \mathbb{F}_2^n \mid \mathrm{d}_H(\mathbf{v}, \mathbf{w}) \leq r\}.$$

Exemple 9 :
Avec $n = 3$ et $\mathbf{v} = \texttt{010}$, on a

1. $\mathcal{B}(\mathbf{v}, r) = \{\texttt{010}\}$ si $0 \leq r < 1$.
2. $\mathcal{B}(\mathbf{v}, r) = \{\texttt{010}, \texttt{110}, \texttt{000}, \texttt{011}\}$ si $1 \leq r < 2$.
3. $\mathcal{B}(\mathbf{v}, r) = \{\texttt{010}, \texttt{110}, \texttt{000}, \texttt{011}, \texttt{100}, \texttt{111}, \texttt{001}\} = \mathbb{F}_2^3 \setminus \{\texttt{101}\}$ si $2 \leq r < 3$.
4. $\mathcal{B}(\mathbf{v}, r) = \mathbb{F}_2^3$ si $3 \leq r$.

4.5.4 Capacité de correction d'un code

Un code est *au moins t-correcteur* s'il est possible de corriger correctement tout mot ayant subi une erreur de poids inférieure ou égale à t. Autrement dit, si

$$\forall \mathbf{v} \in C, \forall \mathbf{e} \in \mathbb{F}_2^n, 0 \leq \omega_H(\mathbf{e}) \leq t \Rightarrow \mathcal{B}(\mathbf{v} \oplus \mathbf{e}, t) \cap C = \{\mathbf{v}\}.$$

Un code est *exactement t-correcteur* s'il est au moins t-correcteur mais pas $t+1$-correcteur.

Exemple 10 :

1. $C = \mathrm{Parite}(k)$ n'est pas au moins 1-correcteur.
2. $C = \mathrm{Rep}(k, p)$ est $\lfloor \frac{p-1}{2} \rfloor$-correcteur.
3. Le code $C = \{\texttt{0000}, \texttt{0111}, \texttt{1010}, \texttt{1111}\}$ est 0-correcteur.
4. Le code $C = \{\texttt{00000}, \texttt{01101}, \texttt{10110}, \texttt{11011}\}$ est exactement 1-correcteur.

4.6 Distance minimale d'un code

La *distance minimale* d'un code C est la plus petite distance séparant deux mots du code. On la note $\mathrm{d}_{min}(C)$.

$$\mathrm{d}_{min}(C) = \min\{\mathrm{d}_H(\mathbf{v}, \mathbf{w}) \mid \mathbf{v}, \mathbf{w} \in C,\ \mathbf{v} \neq \mathbf{w}\}.$$

Théorème 4.3. *Soit C un code et t un entier naturel.*

1. *C est au moins t-détecteur si et seulement si*

$$t < \mathrm{d}_{min}(C).$$

2. *C est au moins t-correcteur si et seulement si*

$$2t + 1 \leq \mathrm{d}_{min}(C),$$

autrement dit si et seulement si

$$t \leq \left\lfloor \frac{\mathrm{d}_{min}(C) - 1}{2} \right\rfloor.$$

Exemple 11 :

1. $\mathrm{d}_{min}(\mathrm{Parite}(k)) = 2$. On retrouve le fait que ce code est exactement 1-détecteur et 0-correcteur.
2. $\mathrm{d}_{min}(\mathrm{Rep}(k,p)) = p$. Ce code est $p-1$-detecteur et $\lfloor \frac{p-1}{2} \rfloor$-correcteur.
3. La distance minimale du code $C = \{\mathtt{0000}, \mathtt{0111}, \mathtt{1010}, \mathtt{1111}\}$ est $\mathrm{d}_{min}(C) = 2$. Ce code est exactement 0-détecteur et 0-correcteur.
4. La distance minimale du code $C = \{\mathtt{00000}, \mathtt{01101}, \mathtt{10110}, \mathtt{11011}\}$ est $\mathrm{d}_{min}(C) = 3$. Il est exactement 2-détecteur et 1-correcteur.

4.7 Codage linéaire

Dans le domaine des codages détecteurs/correcteurs d'erreurs, les codages linéaires se distinguent de par la facilité qu'ils offrent pour les opérations de codage et de détection d'erreurs. Certains d'entre eux permettent en outre un décodage aisé pour la correction d'erreurs.

4.7.1 Codage linéaire

On dit d'un codage qu'il est *linéaire* si l'application de codage

$$\mathbf{c} \ : \ \mathbb{F}_2^k \longrightarrow \mathbb{F}_2^n$$

est une application linéaire (injective), autrement dit si pour tous mots $\mathbf{u}$ et $\mathbf{u}'$ de $\mathbb{F}_2^k$ on a

$$\mathbf{c}(\mathbf{u} \oplus \mathbf{u}') = \mathbf{c}(\mathbf{u}) \oplus \mathbf{c}(\mathbf{u}').$$

Remarque : En algèbre linéaire, une application linéaire est une application qui préserve les combinaisons linéaires, autrement dit telle que pour tous scalaires λ et μ et tous vecteurs $\mathbf{u}$ et $\mathbf{u}'$ on a

$$f(\lambda\mathbf{u} + \mu\mathbf{u}') = \lambda f(\mathbf{u}) + \mu f(\mathbf{u}').$$

Le corps de base des espaces vectoriels $\mathbb{F}_2^n$ étant le corps à deux éléments $\mathbb{F}_2$, les seules valeurs que peuvent prendre les scalaires λ et μ sont 0 et 1. La définition de codage linéaire donnée ci-dessus correspond donc bien à la notion générale d'application linéaire. Les notions d'images, noyaux et matrices s'appliquent donc aux codages linéaires.

4.7.2 Code linéaire

Le code associé à un codage linéaire étant l'image de $\mathbb{F}_2^k$ par une application linéaire, l'algèbre linéaire nous enseigne qu'il est donc un sous-espace vectoriel de dimension k de $\mathbb{F}_2^n$.

Un code linéaire de longueur n, de dimension k et de distance minimale d est appelé un $[n,k,d]$-code linéaire.

Exemple 12 :

1. Le codage par ajout d'un bit de parité est une application linéaire de $\mathbb{F}_2^k$ dans $\mathbb{F}_2^n$ avec $n = k+1$. C'est donc un codage linéaire, et le code Parite(k) est un $[k+1,k,2]$-code linéaire.

2. Le codage par répétition p fois d'un mot de $\mathbb{F}_2^k$ est un codage linéaire dans $\mathbb{F}_2^n$ avec $n = p \cdot k$. Le code associé $\mathrm{Rep}(k, p)$ est un $[pk, k, p]$-code linéaire.

3. Le codage des mots de longueur 2 en des mots de longueur 5 défini par

$$\begin{aligned} \mathbf{c} \ : \ & \mathbb{F}_2^2 \to \mathbb{F}_2^5 \\ & \texttt{00} \mapsto \mathbf{c}(\texttt{00}) = \texttt{00000} \\ & \texttt{01} \mapsto \mathbf{c}(\texttt{01}) = \texttt{01101} \\ & \texttt{10} \mapsto \mathbf{c}(\texttt{10}) = \texttt{10110} \\ & \texttt{11} \mapsto \mathbf{c}(\texttt{11}) = \texttt{11011} \end{aligned}$$

est un codage linéaire dont le code associé $C = \{\texttt{00000}, \texttt{01101}, \texttt{10110}, \texttt{11011}\}$ est un $[5, 2, 3]$-code linéaire.

4.7.3 Distance minimale d'un code linéaire

La distance minimale d'un code linéaire se caractérise comme étant le plus petit poids d'un de ses mots non nuls.

Proposition 4.2. *Soit C un $[n, k, d]$-code linéaire. Alors*

$$d = \min\{\omega_H(\mathbf{v}) \mid \mathbf{v} \in C, \mathbf{v} \neq \mathbf{0}\}.$$

Cette propriété des codes linéaires permet de réduire le nombre de calculs à effectuer pour déterminer leur distance minimale. En effet, déterminer la distance minimale d'un code en général nécessite de calculer toutes les distances séparant deux mots du code. Si le code contient K mots, il s'agit donc de calculer $\frac{K(K-1)}{2}$ distances. Pour un code linéaire de dimension k, on a $K = 2^k$ et le nombre de distances à calculer s'élèvent alors à $2^{2k-1} - 2^{k-1} \sim \frac{4^k}{2}$. En revanche, déterminer la distance minimale d'un code linéaire de dimension k par la recherche du plus petit poids de ses mots non nuls nécessite 2^{k-1} calculs seulement, ce qui représente un gain appréciable par rapport à l'autre méthode. Cependant, la quantité de calculs à fournir augmente encore exponentiellement avec la dimension du code [2].

4.7.4 Matrice génératrice

Comme toutes les applications linéaires, un codage linéaire peut être entièrement défini par l'image d'une base de l'espace de départ.

Une base naturelle des espaces vectoriels de mots binaires $\mathbb{F}_2^n$ est celle constituée des mots de poids 1.

Soit $\mathbf{c}$ un codage linéaire de $\mathbb{F}_2^k$ dans $\mathbb{F}_2^n$. Ce codage est entièrement défini par les mots $\mathbf{v_i}$ de $\mathbb{F}_2^n$ images des mots $\mathbf{u_i}$ de poids 1 de $\mathbb{F}_2^k$.

Exemple 13 :

2. Il est prouvé que le problème de la détermination de la distance minimale d'un code est un problème qui fait partie d'une classe générale de problèmes informatiques considérés comme difficiles : les problèmes NP-complets

1. La matrice génératrice du codage par ajout d'un bit de parité aux mots de longueur k est une matrice $k \times k+1$ dont les k premières colonnes forment la matrice identité, et la dernière colonne est constituée de 1.

$$G = \begin{pmatrix} 1 & 0 & \dots & 0 & 1 \\ 0 & 1 & \ddots & \vdots & \vdots \\ \vdots & \ddots & \ddots & 0 & \vdots \\ 0 & \dots & 0 & 1 & 1 \end{pmatrix}.$$

2. La matrice génératrice du codage par répétition p fois des mots de k bits est une matrice $k \times pk$ constituée de p blocs identité $k \times k$.

$$G = \begin{pmatrix} I_k & I_k & \dots & I_k \end{pmatrix}.$$

3. La matrice du troisième codage de l'exemple 12 est une matrice 2×5.

$$G = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \end{pmatrix}.$$

La représentation d'un codage linéaire par sa matrice génératrice offre une méthode efficace et concise pour représenter les 2^k mots du code. L'opération de codage d'un mot $\mathbf{u} \in \mathbb{F}_2^k$ est une simple multiplication de ce mot (considéré comme un vecteur ligne) par la matrice génératrice du codage :

$$\mathbf{c}(\mathbf{u}) = \mathbf{u} \cdot G.$$

Cette multiplication revient à additionner (modulo 2) les lignes de la matrice G correspondant aux bits non nuls du mot $\mathbf{u}$.

Matrice génératrice des codages linéaires systématiques Pour les codages linéaires qui sont de surcroît systématiques (cf page 74), la matrice génératrice a une forme particulière puisqu'elle est constituée d'un bloc identité $k \times k$ et de $n-k$ colonnes supplémentaires pour l'ajout des bits de contrôle. Les trois exemples de codage ci-dessus sont systématiques.

4.7.5 Matrice de contrôle

La matrice génératrice d'un codage linéaire étant une matrice $k \times n$ de rang k, il existe des matrices H $(n-k) \times n$ de rang $n-k$ telles que

$$G \cdot {}^t H = 0,$$

où ${}^t H$ désigne la transposée de la matrice H.

Exemple 14 :

1. Pour le codage de parité, on peut prendre pour matrice H la matrice $1 \times (k+1)$ dont tous les termes valent 1.

$$H = \begin{pmatrix} 1 & 1 & \dots & 1 \end{pmatrix}.$$

En effet, cette matrice est bien de rang $n-k=1$ et on a bien

$$G \cdot {}^t H = 0.$$

2. Pour le codage par répétition 3 fois des mots de 2 bits, on peut prendre la matrice 4×6

$$H = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

qui est bien de rang $n - k = 4$ et dont on peut vérifier que

$$G \cdot^t H = 0.$$

3. Pour le troisième codage de l'exemple 12, on peut prendre la matrice 3×5

$$H = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix}.$$

En effet cette matrice est bien de rang $5 - 2 = 3$, et on vérifie aisément que

$$G \cdot^t H = 0.$$

De telles matrices H sont appelées *matrices de contrôle*. La justification de cette dénomination réside dans le théorème qui suit.

Théorème 4.4. *Soit C un $[n, k]$-code linéaire, H une matrice de contrôle de ce code et $\mathbf{v} \in \mathbb{F}_2^n$. Alors*

$$\mathbf{v} \in C \iff \mathbf{v} \cdot^t H = \mathbf{0}.$$

Le mot $\mathbf{v} \cdot^t H$ est appelé *syndrome* du mot $\mathbf{v}$ que l'on note $s(\mathbf{v})$. C'est un mot de longueur $n - k$.

Ce théorème signifie que pour détecter qu'un mot $\mathbf{v}'$ est erroné, il suffit de calculer le syndrome du mot $\mathbf{v}'$ et de vérifier qu'il est bien nul. Si c'est le cas, le mot $\mathbf{v}'$ est dans le code, aucune erreur n'est détectée. Si ce n'est pas le cas, le mot $\mathbf{v}'$ n'est pas dans le code, une erreur est détectée et on peut le cas échéant tenter de le corriger. C'est évidemment une méthode de détection particulièrement efficace par rapport à celle consistant à comparer le mot $\mathbf{v}'$ à chacun des 2^k mots du code.

Exemple 15 :
Avec le troisième codage de l'exemple 12,

1. considérons le mot $\mathbf{v}' = \texttt{01101}$ et calculons son syndrome :

$$s(\mathbf{v}') = \texttt{01101} \cdot^t H = \texttt{000},$$

le syndrome est nul, le mot $\mathbf{v}'$ est dans le code ;

2. considérons le mot $\mathbf{v}' = \texttt{10010}$, son syndrome vaut

$$s(\mathbf{v}') = \texttt{10010} \cdot^t H = \texttt{100},$$

et n'est pas nul : le mot $\mathbf{v}'$ n'est pas dans le code.

Théorème 4.5. *Un codage linéaire est au moins 1-détecteur si et seulement si une matrice de contrôle ne possède aucune colonne nulle.*

Plus généralement, avec $1 \leq t \leq n$ il est au moins t-détecteur si et seulement si pour tout entier $1 \leq p \leq t$ les sommes de p colonnes d'une matrice de contrôle ne sont pas nulles.

Démonstration. Soit $\mathbf{c} : \mathbb{F}_2^k \to \mathbb{F}_2^n$ un codage linéaire, C son code associé, et H une matrice de contrôle. Soit $\mathbf{v} \in C$ et $\mathbf{e} \in \mathbb{F}_2^n$ un mot de poids compris entre 1 et t. Posons $\mathbf{v}' = \mathbf{v} \oplus \mathbf{e}$. Le mot $\mathbf{v}'$ est détecté comme erroné si et seulement s'il n'est pas dans le code C. Or

$$\begin{aligned} \mathbf{v}' \notin C &\Leftrightarrow s(\mathbf{v}') \neq \mathbf{0} \\ &\Leftrightarrow (\mathbf{v} \oplus \mathbf{e}) \cdot {}^t H \neq \mathbf{0} \\ &\Leftrightarrow \mathbf{v} \cdot {}^t H \oplus \mathbf{e} \cdot {}^t H \neq \mathbf{0} \\ &\Leftrightarrow \mathbf{e} \cdot {}^t H \neq \mathbf{0} \neq \mathbf{0} \end{aligned}$$

Or $\mathbf{e} \cdot {}^t H$ est la somme des lignes de la matrice ${}^t H$ correspondant aux bits non nuls de $\mathbf{e}$. □

Matrice de contrôle des codages linéaires systématiques Pour un codage linéaire systématique dont la matrice génératice peut donc s'écrire

$$G = \begin{pmatrix} I_k & A \end{pmatrix}$$

où I_k est la matrice Identité $k \times k$ et A une matrice $k \times (n-k)$, la matrice

$$H = \begin{pmatrix} {}^t A & I_{n-k} \end{pmatrix} \iff {}^t H = \begin{pmatrix} A \\ I_{n-k} \end{pmatrix}$$

est une matrice de contrôle.

H est évidemment une matrice $(n-k) \times n$ de rang $n-k$, et on vérifie aisément que

$$G \cdot {}^t H = A \oplus A = 0.$$

4.7.6 Correction par tableau

Avec tout $[n,k]$-code linéaire C, il est possible de partitionner l'ensemble des mots de $\mathbb{F}_2^n$ en sous-ensembles obtenus en ajoutant aux mots de C des mots de poids croissants. On peut présenter cette partition sous forme d'un tableau comprenant 2^{n-k} lignes, chacune des lignes étant constituée des mots d'un ensemble de la forme

$$\mathbf{e} \oplus C = \{\mathbf{e} \oplus \mathbf{v} \mid \mathbf{v} \in C\}.$$

Exemple 16 :
Avec le $[5,2]$-code linéaire

$$C = \{\texttt{00000}, \texttt{01101}, \texttt{10110}, \texttt{11011}\}$$

les $2^5 = 32$ mots de $\mathbb{F}_2^5$ sont répartis dans un tableau de 8 lignes et 4 colonnes (cf tableau 4.1).

Chacune des lignes de ce tableau montre les mots d'un ensemble de la forme $\mathbf{e} \oplus C$, le mot $\mathbf{e}$ étant celui de la deuxième colonne, la première colonne donnant le poids de ce mot. La première ligne correspond à $\mathbf{e} = 0$. Ce sont donc les mots du code. Les 5 lignes qui suivent correspondent à des mots $\mathbf{e}$ de poids 1. Ces mots sont donc à distance 1 du code. Les deux dernières lignes

0	00000	10110	01101	11011	
1	10000	00110	11101	01011	$\oplus$10000
1	01000	11110	00101	10011	$\oplus$01000
1	00100	10010	01001	11111	$\oplus$00100
1	00010	10100	01111	11001	$\oplus$00010
1	00001	10111	01100	11010	$\oplus$00001
2	11000	01110	10101	00011	$\oplus$11000
2	10001	00111	11100	01010	$\oplus$10001

TABLE 4.1 – Partition de $\mathbb{F}_2^5$ par le code C. La dernière colonne représente l'ajout qui a été fait aux mots de la première ligne pour obtenir la ligne courante.

contiennent les mots de $\mathbb{F}_2^5$ à distance 2 du code. Aucun mot de $\mathbb{F}_2^5$ n'est à une distance plus grande que 2 du code.

Ce tableau peut servir à corriger des erreurs. Lorsqu'on reçoit un mot $\mathbf{v}'$ on cherche ce mot dans le tableau, et après l'avoir repéré on le corrige en choisissant le mot de la première ligne situé dans la même colonne que $\mathbf{v}'$.

Exemple 17 :
Supposons que $\mathbf{v}' = \texttt{10010}$. Ce mot est situé dans la 4-ème ligne du tableau 4.1. On le corrige en le mot du code $\mathbf{v} = \texttt{10110}$.

Malheureusement cette technique devient très vite impraticable lorsque n devient grand.

4.8 Codage de Hamming

Un codage de Hamming est un codage correcteur linéaire. L'intérêt des codages de Hamming réside notamment dans leur rendement : il n'est pas possible de trouver de codage ayant les mêmes capacités de détection et correction avec un meilleur rendement. Les codages de Hamming ont donc un rendement optimal.

Dans le cas où l'alphabet source est binaire, les codages de paramètre $[2^m - 1,\ 2^m - 1 - m,\ 3]$ sont des codages de Hamming. Leur rendement est de $\frac{2^m-1-m}{2^m-1}$. Dans la suite nous nous intéresserons au cas où $m = 3$.

4.8.1 Un $[7, 4, 3]$-codage linéaire

Considérons le codage linéaire $\mathbf{c} : \mathbb{F}_2^4 \to \mathbb{F}_2^7$ défini par sa matrice génératrice

$$G = \left(\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right).$$

C'est un codage systématique dont une matrice de contrôle est

$$H = \left(\begin{array}{cccc|ccc} 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{array}\right).$$

Aucune colonne de H n'est nulle, le codage est donc au moins 1-détecteur. Toutes les colonnes de H sont différentes, et donc aucune somme de deux colonnes de H n'est nulle. Le codage est donc au moins 2-détecteur. Mais il existe des sommes nulles de trois colonnes (les trois premières par exemple). Le codage n'est donc pas au moins 3-détecteur. Il est donc exactement 2-détecteur et la distance minimale du code C associé est donc égale à 3.

On en déduit que ce codage est exactement 1-correcteur. Par conséquent pour tout mot $\mathbf{v} \in C$ et tout mot $\mathbf{e} \in \mathbb{F}_2^7$ de poids 1, le mot $\mathbf{v}' = \mathbf{v} \oplus \mathbf{e}$ n'est pas dans le code, et $\mathbf{v}$ est le mot du code le plus proche de $\mathbf{v}'$. Comment effectuer la correction, c.-à-d. comment retrouver $\mathbf{v}$ à partir de $\mathbf{v}$?

On peut toujours recourir à la technique du tableau (cf section 4.7.6), mais cette technique nécessite la construction d'un tableau partionnant les 128 mots binaires de longueur 7.

Une autre technique consiste à remarquer que si $\mathbf{e}$ est de poids 1, alors le syndrome de $\mathbf{v}'$ permet de *localiser* l'erreur. En effet

$$s(\mathbf{v}') = s(\mathbf{e}) = \mathbf{e} \cdot {}^t H,$$

et donc le syndrome de $\mathbf{v}'$ est égal à une ligne de ${}^tH$, autrement dit à une colonne de H. Or toutes les colonnes de H sont différentes.

Exemple 18 :
Le mot $\mathbf{v}' = \texttt{1100010}$ a pour syndrome

$$s(\mathbf{v}') = \texttt{111}.$$

Ce syndrome n'est pas nul, le mot $\mathbf{v}'$ n'est donc pas dans le code. De plus le ce syndrome correspond à la quatrième colonne de H, ce qui signifie que le mot du code le plus proche de $\mathbf{v}'$ est le mot

$$\mathbf{v} = \mathbf{v}' \oplus \texttt{0001000} = \texttt{1101010}.$$

Ce codage est un codage de Hamming.

4.9 Exercices

4.9.1 Canal de communication

Exercice 4-1 *Un canal inutile*

Expliquez pourquoi le canal représenté à la figure 4.5 est appelé *canal inutile*.

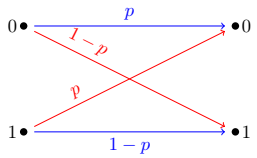


FIGURE 4.5 – Un canal inutile

Exercice 4-2 *Canal vraiment très bruité ?*

Supposons qu'un CBSSM soit vraiment très bruité, c'est–à–dire un canal dans lequel la probabilité p d'erreur lors de la transmission d'un bit soit strictement supérieure à $1/2$.

Question 1 Que dire de ce canal si $p = 1$? Est-il vraiment si bruité ?

Question 2 Que faire si $1/2 < p < 1$?

4.9.2 Détection/correction d'erreurs

Exercice 4-3 *Numéro INSEE*

Le numéro INSEE est un numéro composé de 13 chiffres auquel on ajoute une clé de deux chiffres pour contrôler d'éventuelles erreurs de saisie.

On obtient cette clé

1. en réduisant le numéro N modulo 97 :$r = N \pmod{97}$;
2. en retranchant r à 97 : la clé est $k = 97 - r$.

Par exemple, pour le numéro INSEE $N = 2\,90\,02\,59\,350\,112$ on a

1. $r = N \pmod{97} = 96$;
2. $k = 97 - 96 = 01$.

La clé est un entier compris 1 et 97. On l'écrit en base 10 sur deux chiffres.

Question 1 Comment contrôler la validité d'un couple numéro/clé (N, k) ?

Question 2 Vérifiez que l'ajout de cette clé au numéro INSEE permet de détecter toutes les erreurs simples (modification d'un chiffre, permutation de deux chiffres consécutifs).

Question 3 Permet-il de détecter davantage d'erreurs ?

Exercice 4-4 *Codage de parité à deux dimensions*

Voici une amélioration du codage de parité : on découpe le message à transmettre en blocs de $k = k_1 \times k_2$ bits, et on code séparément chacun de ces blocs. Pour cela

1. on dispose le bloc à coder en k_1 lignes de longueur k_2,
2. on ajoute un bit de parité à la fin de chacune de ces lignes,
3. on ajoute une $k_1 + 1$-ème ligne en faisant le codage de parité des $k_2 + 1$ colonnes formées par les k_1 lignes qui précèdent,
4. puis on forme le mot de longueur $n = (k_1 + 1)(k_2 + 1)$ obtenu en concaténant ces $k_1 + 1$ lignes.

Par exemple, avec $k_1 = 5$ et $k_2 = 4$, pour transmettre le bloc $\mathbf{u} =$ `00101110010100010110`, on construit les lignes :

```
0010 1
1110 1
0101 0
0001 1
0110 0

1110 1
```

et on transmet le mot $\mathbf{c}(\mathbf{u}) = \texttt{00101111010101000011011001110 1}$.

Étudiez les propriétés de détection et correction d'erreurs de ce codage.

Exercice 4-5 *Capacité de correction*

Question 1 Calculez la capacité de détection et de correction des codes binaires suivants :

1. $C = \{\texttt{11100}, \texttt{01001}, \texttt{10010}, \texttt{00111}\}$
2. $C = \{\texttt{11111111}, \texttt{11001100}, \texttt{10101010}, \texttt{00000000}\}$.

Question 2 Soit d la capacité de détection d'erreurs d'un codage. Déduisez-en sa capacité de correction.

Exercice 4-6 *Longueur minimale*

On veut un codage binaire au moins 1-correcteur pour coder 16 mots. Quelle est la longueur minimale des mots du code ?

Exercice 4-7 *Codage de Hamming*

Le mot $\mathbf{v}' = \texttt{1100110}$ a été reçu. En sachant que le mot émis a été codé avec le codage de Hamming $[7, 4, 3]$ (cf section 4.8.1), et en faisant l'hypothèse qu'au plus un bit est erroné, quel est le mot qui a été émis ?

Annexe A

Autour du binaire

A.1 Puissances de 2

n	2^n	Ordre de grandeur
0	1	
1	2	
2	4	
3	8	
4	16	
5	32	
6	64	
7	128	
8	256	
9	512	
10	1 024	10^3 (kilo)
11	2 048	
12	4 096	
13	8 192	
14	16 384	
15	32 768	
16	65 536	
17	131 072	
18	262 144	
19	524 288	
20	1 048 576	10^6 (méga)
30	1 073 741 824	10^9 (giga)
40	1 099 511 627 776	10^{12} (téra)
50	1 125 899 906 842 624	10^{15} (péta)

TABLE A.1 – Premières puissances de 2

Remarque : Longtemps les préfixes *kilo*, *méga*, ..., appliqués aux octets pour mesurer des tailles de mémoire désignaient des puissances de 2. Or pour toute autre mesure dans le système

international, ces mêmes préfixes désignent des puissances de 10. Pour rémédier à cela, la commission électronique internationale (IEC) a spécifié de nouveaux préfixes pour les puissances de 2 (cf table A.2).

Préfixe décimal (SI)			Préfixe binaire (IEC)		
Nom	Puissance	Abbréviation	Nom	Puissance	Abbréviation
kilooctet	10^3	Ko	kibioctet	2^{10}	Kio
mégaoctet	10^6	Mo	mébioctet	2^{20}	Mio
gigaoctet	10^9	Go	gibioctet	2^{30}	Gio
téraoctet	10^{12}	To	tébioctet	2^{40}	Tio
pétaoctet	10^{15}	Po	pébioctet	2^{50}	Pio

TABLE A.2 – Préfixes des puissances binaires et décimales appliqués aux octets

A.2 Logarithme binaire

Par définition, le logarithme binaire d'un réel positif x est

$$\log_2 x = \frac{\ln x}{\ln 2},$$

où $\ln x$ désigne le logarithme népérien (ou naturel) du réel x.

En particulier, la fonction $x \mapsto \log_2 x$ est continue, dérivable et strictement croissante sur l'intervalle $]0, +\infty[$, et on a

$$\begin{aligned} \log_2 1 &= 0 \\ \log_2 2 &= 1 \\ \lim_{x\to+\infty} \log_2 x &= +\infty \\ \lim_{x\to 0} \log_2 x &= -\infty. \end{aligned}$$

Les relations algébriques valables pour la fonction ln le sont aussi pour $\log_2$: pour tous réels $x, y > 0$, et tout réel α on a

$$\begin{aligned} \log_2 xy &= \log_2 x + \log_2 y \\ \log_2 \frac{x}{y} &= \log_2 x - \log_2 y \\ \log_2 x^\alpha &= \alpha \log_2 x. \end{aligned}$$

En particulier, on a

$$\log_2 x = n \in \mathbb{Z} \iff x = 2^n.$$

Annexe B

Quelques codages

B.1 Le Morse

La table B.1 donne le codage en Morse des 26 lettres de l'alphabet latin, ainsi que des 10 chiffres décimaux. En réalité, le Morse peut aussi coder les principaux symboles de ponctuation, ainsi que certains caractères accentués.

a	·–	j	·–––	s	···	1	·————
b	–···	k	–·–	t	–	2	··–––
c	–·–·	l	·–··	u	··–	3	···––
d	–··	m	––	v	···–	4	····–
e	·	n	–·	w	·––	5	·····
f	··–·	o	–––	x	–··–	6	–····
g	––·	p	·––·	y	–·––	7	––···
h	····	q	––·–	z	––··	8	–––··
i	··	r	·–·	0	–––––	9	––––·

TABLE B.1 – Le codage Morse

Remarque : Contrairement aux apparences, l'alphabet utilisé par le Morse n'est pas limité aux deux seuls symboles point · et tiret – . Un troisième symbole est nécessaire représentant le silence [1] (voir page 37).

B.2 Le code Baudot

Codage créé en 1874 par Émile Baudot, puis modifié et utilisé pour le réseau télégraphique commuté, connu aussi sous le nom CCITn° 2, depuis 1917. Il permet de coder 57 caractères avec des codes de 5 bits. Cf table B.2.

1. N'oublions pas que le Morse est non pas un code écrit mais un code sonore.

B.3 Le codage ASCII

Code initié en 1963 puis complété en 1967, le code ASCII-1967 est devenu une norme ISO-646 en 1983. Il permet de coder 128 caractères, à l'origine sur 7 bits, puis sur huit bits. Cf table B.3.

B.4 Le codage ISO-8859-1

b_4 b_3 b_2 b_1	b_8 0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	
	b_7 0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1	1	
	b_6 0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	
	b_5 0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	
	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15		
0 0 0 0	00			SP	0	@	P	`	p			NBSP	°	À	Ð	à	ð	0
0 0 0 1	01			!	1	A	Q	a	q			¡	±	Á	Ñ	á	ñ	1
0 0 1 0	02			"	2	B	R	b	r			¢	²	Â	Ò	â	ò	2
0 0 1 1	03			#	3	C	S	c	s			£	³	Ã	Ó	ã	ó	3
0 1 0 0	04			$	4	D	T	d	t			¤	´	Ä	Ô	ä	ô	4
0 1 0 1	05			%	5	E	U	e	u			¥	µ	Å	Õ	å	õ	5
0 1 1 0	06			&	6	F	V	f	v			¦	¶	Æ	Ö	æ	ö	6
0 1 1 1	07			'	7	G	W	g	w			§	·	Ç	×	ç	÷	7
1 0 0 0	08			(	8	H	X	h	x			¨	¸	È	Ø	è	ø	8
1 0 0 1	09			)	9	I	Y	i	y			©	¹	É	Ù	é	ù	9
1 0 1 0	10			*	:	J	Z	j	z			ª	º	Ê	Ú	ê	ú	A
1 0 1 1	11			+	;	K	[	k	{			«	»	Ë	Û	ë	û	B
1 1 0 0	12			,	<	L	\	l	|			¬	¼	Ì	Ü	ì	ü	C
1 1 0 1	13			-	=	M	]	m	}			SHY	½	Í	Ý	í	ý	D
1 1 1 0	14			.	>	N	^	n	~			®	¾	Î	Þ	î	þ	E
1 1 1 1	15			/	?	O	_	o				¯	¿	Ï	ß	ï	ÿ	F
			0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	hex

ftp://std.dkuug.dk/JTC1/sc2/wg3/docs/n411.pdf

Code				Caractères en	
binaire	octal	hexadécimal	décimal	mode lettres	mode chiffres
00000	00	00	0	Rien (NUL)	
00001	01	01	1	T	5
00010	02	02	2	Retour chariot (CR)	
00011	03	03	3	O	9
00100	04	04	4	Espace (SP)	=
00101	05	05	5	H	£
00110	06	06	6	N	,
00111	07	07	7	M	.
01000	10	08	8	Saut de ligne (LF)	
01001	11	09	9	L	)
01010	12	0A	10	R	4
01011	13	0B	11	G	&
01100	14	0C	12	I	8
01101	15	0D	13	P	0
01110	16	0E	14	C	:
01111	17	0F	15	V	;

Code				Caractères en	
binaire	octal	hexadécimal	décimal	mode lettres	mode chiffres
10000	20	10	16	E	3
10001	21	11	17	Z	"
10010	22	12	18	D	$
10011	23	13	19	B	?
10100	24	14	20	S	Sonnerie (BEL)
10101	25	15	21	Y	6
10110	26	16	22	F	!
10111	27	17	23	X	/
11000	30	18	24	A	-
11001	31	19	25	W	2
11010	32	1A	26	J	’
11011	33	1B	27	Active le mode chiffres	
11100	34	1C	28	U	7
11101	35	1D	29	Q	1
11110	36	1E	30	K	(
11111	37	1F	31	Active le mode lettres	

Table B.2 – Codage Baudot (source Wikipedia)

	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	ESP	!	"	#	$	%	&	’	(	)	*	+	,	-	.	/
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

TABLE B.3 – Table du codage ASCII (ISO-646)

Table des matières

(version du 11 juillet 2019.)