

## Module 2 : Information et Communication

### Leçon 2.4 : Compression de données (2ème partie)

O. Lévêque – Faculté Informatique et Communications

# Plan

- **Plan détaillé de ces deux leçons :**
- **La semaine dernière:**
  - notion d'entropie
  - compression sans pertes
  - algorithme de Shannon-Fano
- **Aujourd'hui**
  - algorithme de Shannon-Fano (bis)
  - algorithme de Huffman
  - analyse de performance – théorème de Shannon
  - compression avec pertes

## Compression sans pertes : rappel

L'entropie  $H(X)$  d'une séquence de lettres : 
$$H(X) = p_1 \log_2 \left( \frac{1}{p_1} \right) + \dots + p_n \log_2 \left( \frac{1}{p_n} \right)$$

Mesure le *désordre* / *la variété* / l'« *information* » contenue dans cette séquence :

$n$  = taille de l'alphabet

$p_j$  = probabilité d'apparition d'une lettre

$\log_2(1/p_j)$  = « nombre » de questions binaires nécessaires pour deviner la  $j^e$  lettre

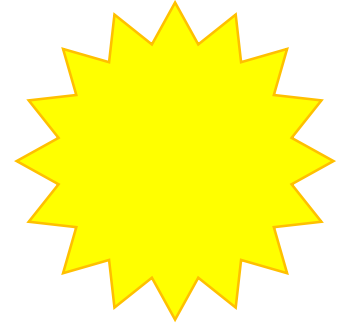
L'*algorithme de Shannon-Fano* permet de représenter une séquence de lettres à l'aide de, *au mieux*,  $H(X)$  bits par lettre :

**Règle n° 1** : le nombre de bits attribués à chaque lettre est égal au nombre de questions nécessaires pour la deviner.

**Règle n° 2** : les bits 0 ou 1 sont attribués en fonction des réponses obtenues aux questions.

## Algorithme de Shannon-Fano (bis)

**Remarque :** La séquence prise en exemple la semaine dernière (IL FAIT BEAU A IBIZA) est particulière, car la probabilité d'apparition  $p$  de chaque lettre est une puissance inverse de 2 (c'est-à-dire  $1/4$ ,  $1/8$  ou  $1/16$ ).



Donc pour chaque lettre,  $\log_2(1/p)$  est un nombre entier, qui correspond exactement au nombre de questions nécessaires pour la deviner.

En pratique, ce n'est bien sûr pas toujours le cas...

# Algorithme de Shannon-Fano (bis)

- Considérons par exemple cette autre séquence (**12 lettres** au total, sans les espaces) :

- JE PARS A PARIS

- Comment procéder pour représenter cette séquence avec un nombre minimum de bits par lettre ?
- A nouveau, reprenons notre jeu des questions et construisons le même tableau qu'avant, en classant les lettres par *ordre décroissant du nombre d'apparitions* :

lettre	A	P	R	S	J	E	I
nb d'apparitions	3	2	2	2	1	1	1

On remarque ici qu'il n'est pas possible de diviser l'ensemble des lettres en deux parties de telle façon que le nombre d'apparitions à gauche égale le nombre d'apparitions à droite.



## Algorithme de Shannon-Fano (bis)

lettre	A	P	R	S	J	E	I
nb d'apparitions	3	2	2	2	1	1	1

- Cependant, on peut faire en sorte de *minimiser la différence* entre le nombre d'apparitions à gauche et à droite.
- Dans l'exemple ci-dessus, on a en fait deux possibilités équivalentes pour la première question :
  - soit « est-ce que la lettre est un A ou un P ? »
  - soit « est-ce que la lettre est un A, un P ou un R ? »
- Supposons qu'on choisisse la première possibilité ; le jeu continue comme précédemment, avec cette nouvelle règle un peu plus souple...

# Algorithme de Shannon-Fano (bis)

Ce qui nous amène (par exemple !) au tableau suivant :

lettre	A	P	R	S	J	E	I
nb d'apparitions	3	2	2	2	1	1	1
nb de questions	2	2	3	3	3	4	4

Donc le nombre moyen de questions à poser pour deviner une lettre est :

$$\frac{5}{12} \times 2 + \frac{5}{12} \times 3 + \frac{2}{12} \times 4 = \frac{10 + 15 + 8}{12} = \frac{33}{12} = \frac{11}{4} = 2.75$$

La règle d'attribution des mots de code à chaque lettre est exactement la même qu'avant :

**Règle n° 1 :** le nombre de bits attribués à chaque lettre est égal au nombre de questions nécessaires pour la deviner.

**Règle n° 2 :** les bits 0 ou 1 sont attribués en fonction des réponses obtenues aux questions. Plus précisément, le bit n° j est égal à 1 ou 0 selon que la réponse à la j<sup>e</sup> question était oui ou non.

A	3				
P	2				
R	2				
S	2				
J	1				
E	1				
I	1				

## Algorithme de Shannon-Fano (bis)

Ce qui donne le dictionnaire suivant :

lettre	A	P	R	S	J	E	I
nb d'apparitions	3	2	2	2	1	1	1
nb de questions	2	2	3	3	3	4	4
mot de code	11	10	011	010	001	0001	0000

(remarquer à nouveau qu'aucun mot de code n'est le préfixe d'un autre, *par construction*)

Un calcul en tout point identique au précédent montre que le nombre de bits utilisés pour représenter la séquence

JE PARS A PARIS

au moyen du code ci-dessus est égal à **33**

et le nombre moyen de bits par lettre est donc égal à  $33/12 = 2.75$



## Algorithme de Shannon-Fano (bis)

Pour finir, calculons l'entropie de la séquence :

lettre	A	P	R	S	J	E	I
nb d'apparitions	3	2	2	2	1	1	1
probabilité $p$	1/4	1/6	1/6	1/6	1/12	1/12	1/12

(remarquer que  $\log_2(1/p)$  n'est pas nécessairement un nombre entier)

Le calcul de l'entropie donne :

$$H(X) = \frac{1}{4} \log_2(4) + \frac{3}{6} \log_2(6) + \frac{3}{12} \log_2(12) = \mathbf{2.69} \text{ bit}$$

qui est un peu plus petite que la valeur **2.75** trouvée précédemment.

# Algorithme de Shannon-Fano (bis)

**Une dernière remarque :** Du point de vue du taux de compression (nb moyen de bits/lettre) l'algorithme de Shannon-Fano offre une très bonne performance dans la majorité des cas mais n'est pas garanti être optimal.

*Le taux de compression peut être différent selon les regroupements effectués même en respectant la règle de minimiser la différence entre les groupes*

**Exemple:** soit la séquence de **7** lettres : **ABRAXAS**

Apparitions: 3x A et 1x 4 lettres distinctes

Regroupement : 4 | 3

3	2 bits
1	2 bits
-----	
1	2 bits
1	3 bits
1	3 bits

donc  $5 \times 2 + 2 \times 3 = 16$  bits au total.

Regroupement : 3 | 4

3	1 bit
-----	
1	3 bits
1	3 bits
1	3 bits
1	3 bits

donc  $3 \times 1 + 4 \times 3 = 15$  bits au total.

# De Shannon-Fano à Huffman

- En pratique, on utilise l'algorithme de Huffman qui travaille avec une approche bottom-up de regroupement des nombres d'apparitions:

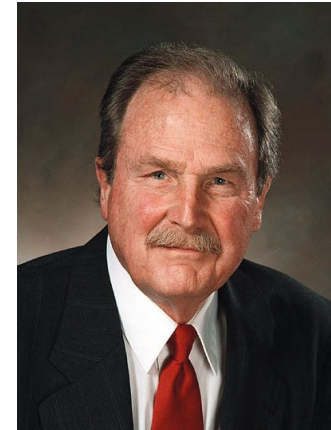
**Initialisation:** ensemble des groupes = (lettre, nb apparitions)

**Faire**

Choisir les 2 groupes avec le plus faible nb d'apparitions

Remplacer ces 2 groupes par un nouveau groupe ayant le total du nombre d'apparitions des 2 groupes

**Tant que** nombre de groupes  $> 1$



David Albert Huffman  
(1925 - 1999)  
ingénieur électricien

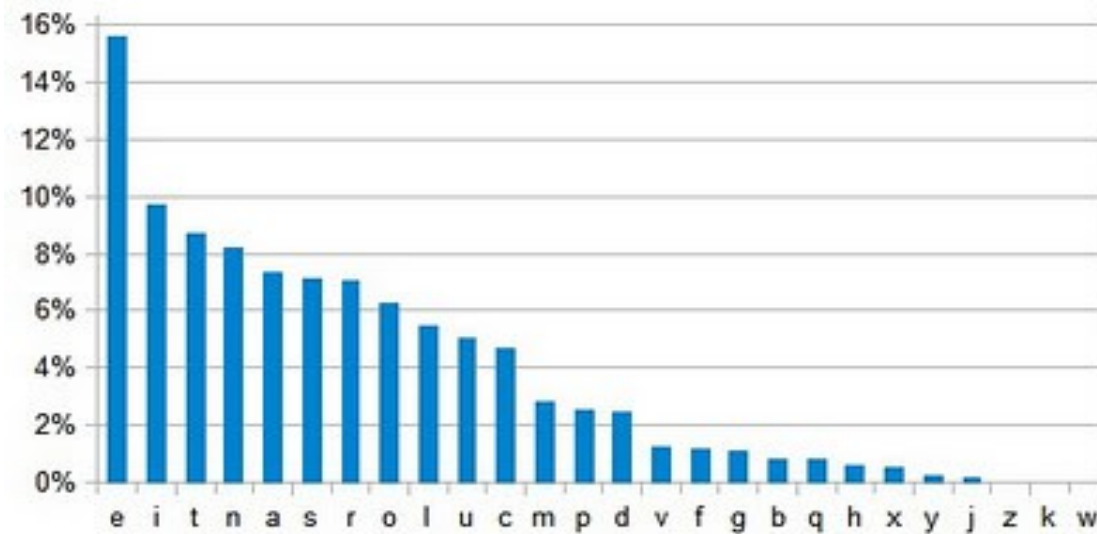
Une performance optimale est obtenue quelle que soit la manière de construire les groupes selon l'algorithme de Huffman.

Exercice: appliquer l'algorithme de Huffman sur JE PARS A PARIS

lettre	A	P	R	S	J	E	I
nb d'apparitions	3	2	2	2	1	1	1

## A propos du code...

- **Question** : Doit-on toujours recréer un code « depuis le début » pour représenter un message ?
- **Réponse** : Non ! On peut simplement utiliser un code basé sur les probabilités d'apparition des lettres dans la langue française :



(tiré de la déclaration universelle des droits de l'homme)

## code optimal ?

- **Il s'agit d'un compromis** : un code peut être optimal pour la distribution d'apparition des lettres pour une langue donnée, et ne pas l'être pour des textes particuliers.
- L'exemple ci-dessous respecte une contrainte qui n'est pas à son avantage pour une compression optimale :

Il prit un roman, il l'ouvrit, il lut; mais il n'y saisissait qu'un imbroglio confus, il butait à tout instant sur un mot dont il ignorait la signification.

# Analyse de performance

## Quelques définitions :

Un *code binaire* est un ensemble  $C$  dont les éléments  $c_1, \dots, c_n$  (également appelés *mots de code*) sont des suites de 0 et de 1 de longueur finie.

Exemple : { 11, 10, 011, 010, 001, 0001, 0000 }

On note  $l_j$  la longueur d'un mot de code  $c_j$ .

Exemple :  $l_5 = 3$

Un code binaire  $C$  est dit *sans préfixe* si aucun mot de code n'est le préfixe d'un autre. Ceci garantit :

- que tous les mots de code sont différents ;
- que tout message formé de ces mots de code peut être décodé au fur et à mesure de la lecture.
- Exemple : avec le code ci-dessus, 1101010011 se lit 11, 010, 10, 011

# Analyse de performance

## Quelques définitions (suite) :

Le code  $C = \{c_1, \dots, c_n\}$  peut être utilisé pour représenter une séquence  $X$  formée avec des lettres tirées d'un alphabet  $A = \{a_1, \dots, a_n\}$ :

chaque lettre  $a_j$  est représentée par le mot de code  $c_j$  de longueur  $l_j$

Exemple :

lettre	A	P	R	S	J	E	I
mot de code	11	10	011	010	001	0001	0000

Si les lettres  $a_1, \dots, a_n$  apparaissent avec des probabilités  $p_1, \dots, p_n$  dans la séquence  $X$ , alors la **longueur moyenne du code** (i.e. le nombre moyen de bits utilisés par lettre) est donnée par

$$L(C) = p_1 l_1 + \dots + p_n l_n$$



# Analyse de performance

**Théorème de Shannon** : *Quel que soit* le code binaire  $C$  sans préfixe utilisé pour représenter une séquence  $X$  donnée, l'inégalité suivante est toujours vérifiée :

$$L(C) \geq H(X)$$

On voit donc également ici apparaître un **seuil** (cf. aussi théorème d'échantillonnage) : en dessous de ce seuil, il n'est pas possible de compresser des données sans faire de perte.

Pour démontrer ce théorème, nous avons besoin de l'inégalité suivante :

**Inégalité de Kraft** : Soit  $C = \{c_1, \dots, c_n\}$  un code binaire sans préfixe.

Alors (rappelons que  $l_j$  est la longueur du mot de code  $c_j$ ).

$$2^{-l_1} + \dots + 2^{-l_n} \leq 1$$

## Démonstration de l'inégalité de Kraft :

Soit  $l_{\max}$  la longueur du mot de code le plus long dans  $C$ .

On peut représenter chaque mot du code  $C$  par un nœud dans un arbre binaire de profondeur  $l_{\max}$ .

On appelle « descendants » d'un mot de code donné les mots de code situés en dessous de celui-ci dans l'arbre.

### Observations :

Au niveau  $l_{\max}$ , il y a  $2^{l_{\max}}$  nœuds en tout.

Le mot de code  $c_j$  a  $2^{l_{\max}-l_j}$  descendants au niveau  $l_{\max}$ .

A cause de l'hypothèse (code sans préfixe), les descendants de mots de code distincts sont tous distincts également.

Donc

$$2^{l_{\max}-l_1} + \dots + 2^{l_{\max}-l_n} \leq 2^{l_{\max}}$$

Longueur code	1	2	3	4 $l_{\max}$		
Nb desc.					$2^{l_{\max}-l_j}$	
						<b>A</b>
						<b>P</b>
						<b>R</b>
						<b>S</b>
						<b>J</b>
						<b>E</b>
						<b>I</b>

En divisant de chaque côté par  $2^{l_{\max}}$ , on obtient l'inégalité de Kraft.

## Démonstration du théorème de Shannon :

Par définition :

$$\begin{aligned} H(X) - L(C) &= \left( p_1 \log_2 \left( \frac{1}{p_1} \right) + \dots + p_n \log_2 \left( \frac{1}{p_n} \right) \right) - (p_1 \ell_1 + \dots + p_n \ell_n) \\ &= p_1 \left( \log_2 \left( \frac{1}{p_1} \right) - \ell_1 \right) + \dots + p_n \left( \log_2 \left( \frac{1}{p_n} \right) - \ell_n \right) \end{aligned}$$

Remarquer que  $-\ell_j = \log_2 (2^{-\ell_j})$  , donc

$$\log_2 \left( \frac{1}{p_j} \right) - \ell_j = \log_2 \left( \frac{1}{p_j} \right) + \log_2 (2^{-\ell_j}) = \log_2 \left( \frac{2^{-\ell_j}}{p_j} \right)$$

d'où

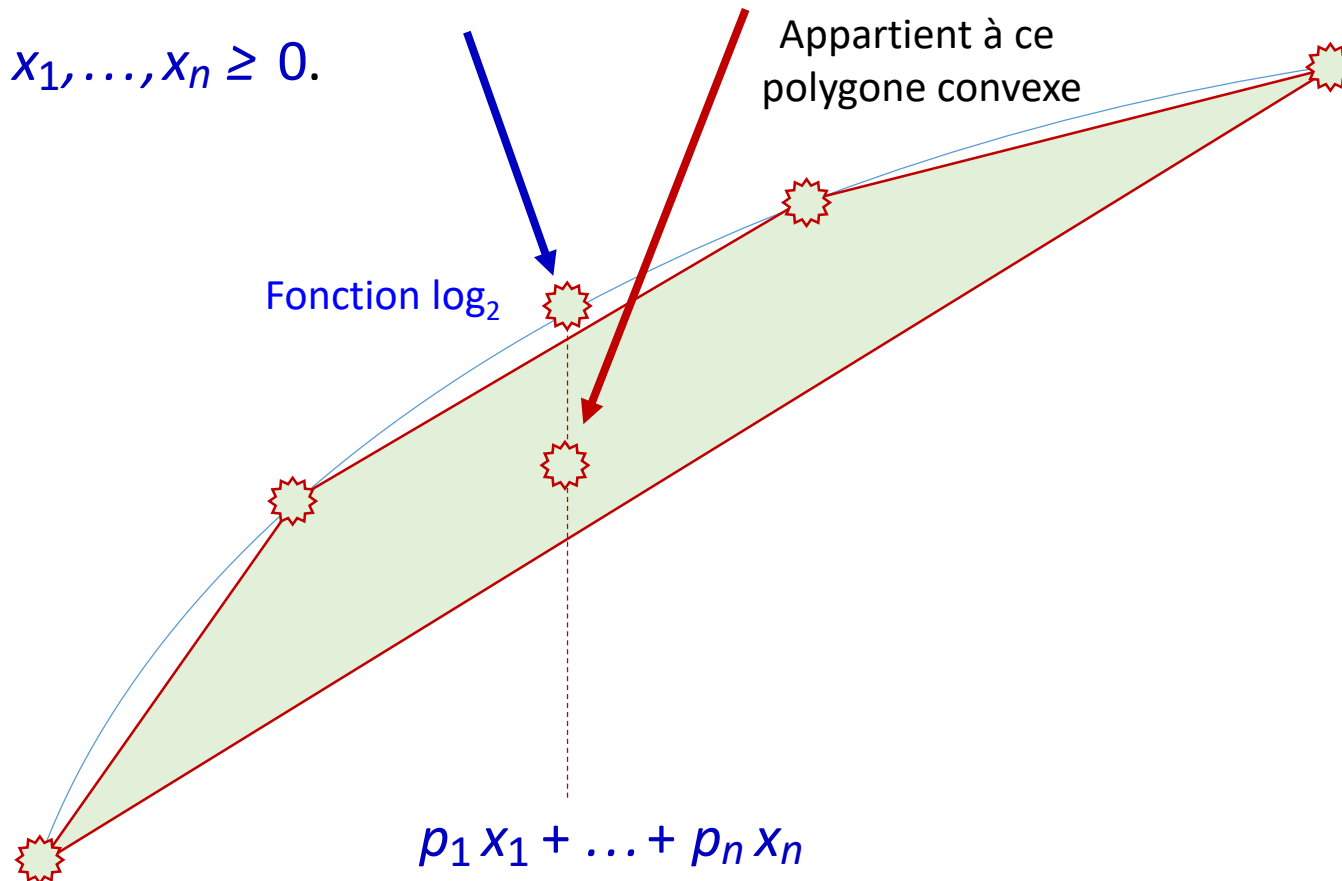
$$H(X) - L(C) = p_1 \log_2 \left( \frac{2^{-\ell_1}}{p_1} \right) + \dots + p_n \log_2 \left( \frac{2^{-\ell_n}}{p_n} \right)$$

# Concavité de la fonction $\log()$

Plus généralement encore, si  $0 \leq p_j \leq 1$  et  $p_1 + \dots + p_n = 1$ , alors

$$\log_2(p_1 x_1 + \dots + p_n x_n) \geq p_1 \log_2(x_1) + \dots + p_n \log_2(x_n)$$

pour tous  $x_1, \dots, x_n \geq 0$ .



## Démonstration du théorème de Shannon (suite) :

En utilisant à nouveau le fait que  $f(x) = \log_2(x)$  est concave, on obtient :

$$\begin{aligned} H(X) - L(C) &= p_1 \log_2 \left( \frac{2^{-l_1}}{p_1} \right) + \dots + p_n \log_2 \left( \frac{2^{-l_n}}{p_n} \right) \\ &\leq \log_2 \left( p_1 \frac{2^{-l_1}}{p_1} + \dots + p_n \frac{2^{-l_n}}{p_n} \right) = \log_2 \left( 2^{-l_1} + \dots + 2^{-l_n} \right) \end{aligned}$$

Par l'inégalité de Kraft :  $2^{-l_1} + \dots + 2^{-l_n} \leq 1$

et le fait que  $f(x) = \log_2(x)$  est croissante, on conclut que

$$H(X) - L(C) \leq \log_2(1) = 0$$

donc  $H(X) \leq L(C)$ , ce qui prouve le théorème de Shannon.

# Théorème de Shannon :

cas particulier où les probabilités  $p$  sont de la forme  $2^{-k}$

IL FAIT BEAU A IBIZA

lettre	I	A	B	L	F	T	E	U	Z
probabilité $p$	$2^{-2}$	$2^{-2}$	$2^{-3}$	$2^{-4}$	$2^{-4}$	$2^{-4}$	$2^{-4}$	$2^{-4}$	$2^{-4}$
nb de questions = nb de bits	2	2	3	4	4	4	4	4	4

Quand  $p_k = 2^{-k}$  la longueur du code est égale à  $k$

$$H(X) - L(C) = p_1 \log_2 \left( \frac{2^{-l_1}}{p_1} \right) + \dots + p_n \log_2 \left( \frac{2^{-l_n}}{p_n} \right)$$

Chaque terme de la somme est de la forme :  $2^{-k} \log_2 (2^{-k} / 2^{-k}) = 2^{-k} \log_2 (1) = 2^{-k} \cdot 0 = 0$

L'entropie  $H(X)$  est égale la longueur moyenne de code  $L(C)$  dans ce cas particulier

# Analyse de performance

On peut montrer également qu'avec l'algorithme de Shannon-Fano,

$$L(C) \leq H(X) + 1 \text{ (bit /lettre)}$$

et donc que la performance de celui-ci est proche de la meilleure performance qu'on puisse espérer.

## Remarques :

L'inégalité ci-dessus est en général pessimiste. On a vu dans l'exemple précédent que  $L(C)$  peut être plus proche de  $H(X)$ , et même égal à  $H(X)$  dans le cas idéal.

La démonstration de cette inégalité est un peu technique, donc nous ne la ferons pas ici.

La performance de l'algorithme de Huffman (qui est optimale) est aussi dans ces bornes.

# Compression avec pertes

Pour compresser des données sans perte (et avec un code sans préfixe), on ne peut donc pas descendre en-dessous de la borne de Shannon.

Si on essaye malgré tout en utilisant un algorithme du type « Shannon-Fano », on court à la catastrophe !

**Exemple** avec le message :

JE PARS A PARIS

Essayons d'utiliser le code suivant :

lettre	A	P	R	S	J	E	I
mot de code	1	0	11	10	01	00	111

Avec un tel code, on n'utilise que **18** bits au total, et pas **33**, mais la représentation binaire du message donne : **0100...**

Le destinataire ne peut donc pas savoir si le message commence par « JE », « JPP », « PAE », « PAPP » ou encore « PSP »...



## Compression avec pertes

Cependant, on est parfois *obligé* de compresser en faisant des pertes :

lorsqu'on désire représenter un nombre réel avec un nombre fixé de bits (l'information comprise dans un nombre réel est « infinie »);

lorsqu'on désire échantillonner un signal dont la plus grande fréquence est « infinie »;

lorsqu'on désire télécharger sur un site web l'intégralité de ses photos de vacances (quelques Gigaoctets pour une centaine de photos avec la plus haute résolution).

Comment procéder dans ce dernier cas ?

D'abord filtrer les hautes fréquences !

## Compression avec pertes : images

*Idée: ça n'est pas un problème de «perdre» ce que l'on ne perçoit pas.  
Exemple: souvent la résolution d'une image est bien plus grande qu'il n'est nécessaire pour apparaître en petit format dans un coin de page web.  
Ensuite tout dépend de la distance à laquelle on regarde l'image.*



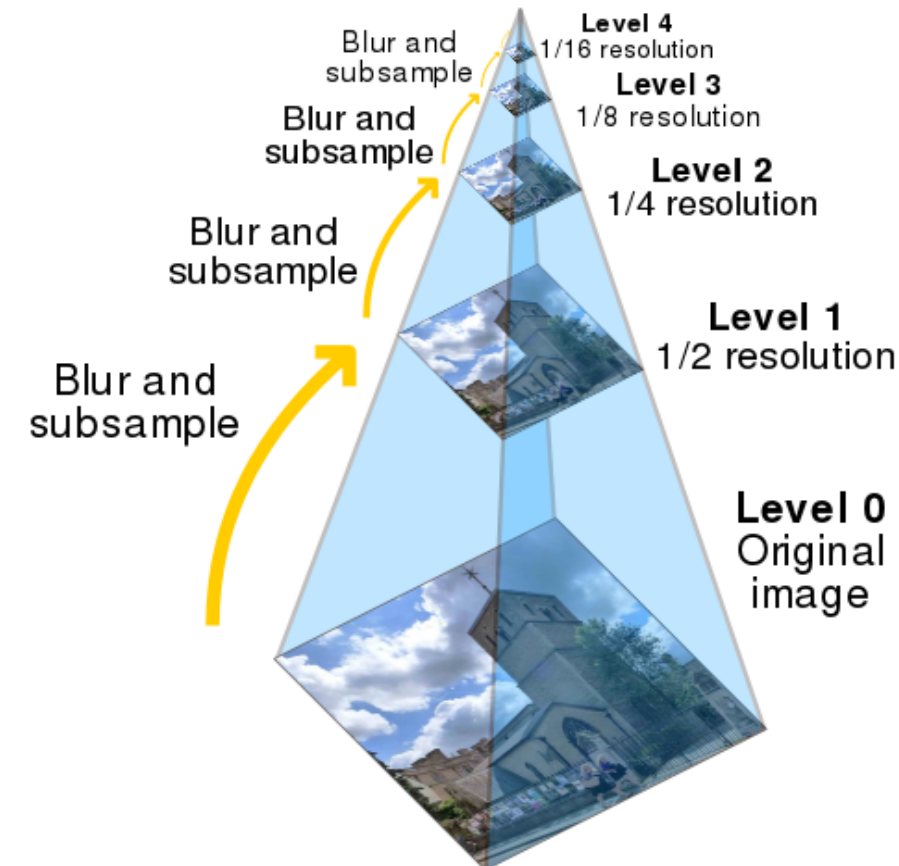
# Compression avec pertes : images

- De la même manière que l'oreille humaine n'est pas capable de percevoir des sons au-delà de  $\sim 20$  kHz, l'œil humain a un pouvoir de résolution d'environ **une minute d'arc =  $1/60 \sim 0.017$  degré**, ce qui veut dire qu'il ne distingue pas :

- des objets de taille inférieure à 1mm situés à 3m de distance
- des pixels de taille inférieure à  $0.2 \times 0.2$ mm sur un écran d'ordinateur (à 50cm de distance).

- Comment filtrer les hautes fréquences (spatiales) d'une image ?
- Une façon simple de faire : réduire la résolution de l'image en moyennant les couleurs sur des zones de plus ou moins grande taille.

☞ filtrage de type pyramide



[https://en.wikipedia.org/wiki/Pyramid\\_\(image\\_processing\)](https://en.wikipedia.org/wiki/Pyramid_(image_processing))

By Cmglee - Own work, CC BY-SA 3.0,  
<https://commons.wikimedia.org/w/index.php?curid=42549151>

# Compression avec pertes : images

On voit apparaître un *compromis* :

plus on utilise des pixels de grande taille, moins on a besoin d'espace-mémoire pour stocker l'image...

mais plus le signal d'origine est déformé : on parle de **distorsion**.

Il existe bien sûr des algorithmes beaucoup plus sophistiqués pour comprimer une image avec pertes :

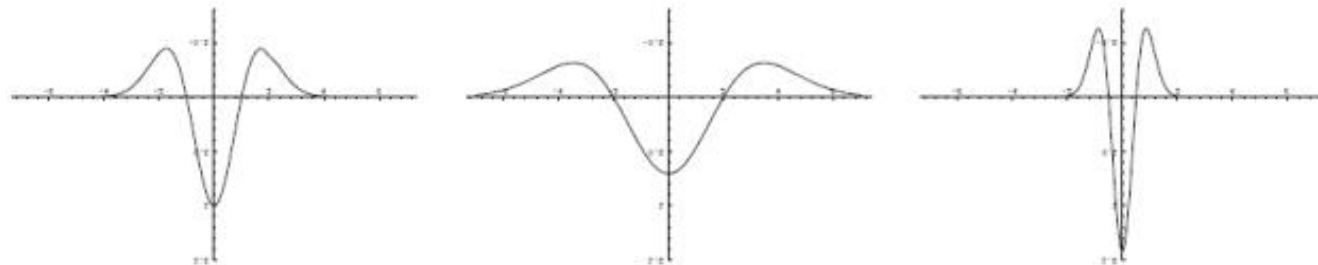
format JPEG :

analyse les fréquences spatiales présentes dans l'image ;

n'en retient que les plus basses ;

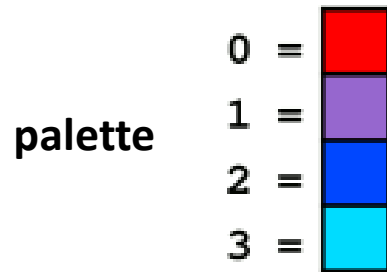
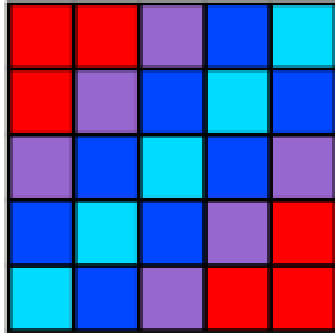
utilise un algorithme de compression sans pertes sur le résultat.

format JPEG 2000 : la même chose, mais avec des *ondelettes*.



*ondelette mexican hat  
à différentes échelles*

# Une alternative pour la compression des images avec peu de couleurs



+ table d'index

0	0	1	2	3
0	1	2	3	2
1	2	3	2	1
2	3	2	1	0
3	2	1	0	0

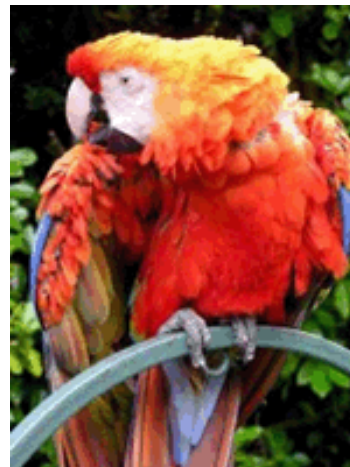
codage sans perte d'une image si l'ensemble des couleurs présentes dans l'image est limité à une palette restreinte: couleurs indexées GIF / PNG

construction d'une **palette** de 4, 16 ou 256 couleurs

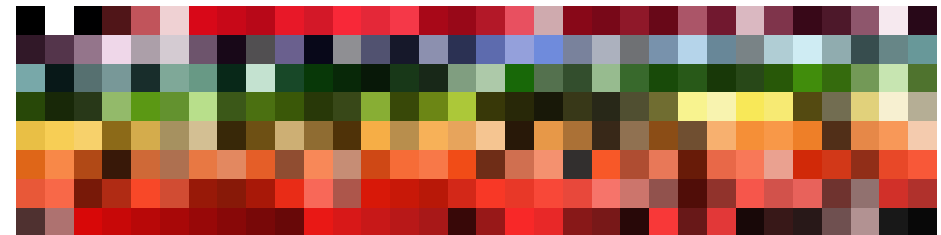
- chaque pixel mémorise seulement l'**index** vers la palette.
- chaque couleur de la **palette** est définie avec 4 octets

pour des images avec:

- un petit nombre de couleurs: dessin technique, BD, manga, comics
- des nuances mais très localisées



Palette de 256 couleurs



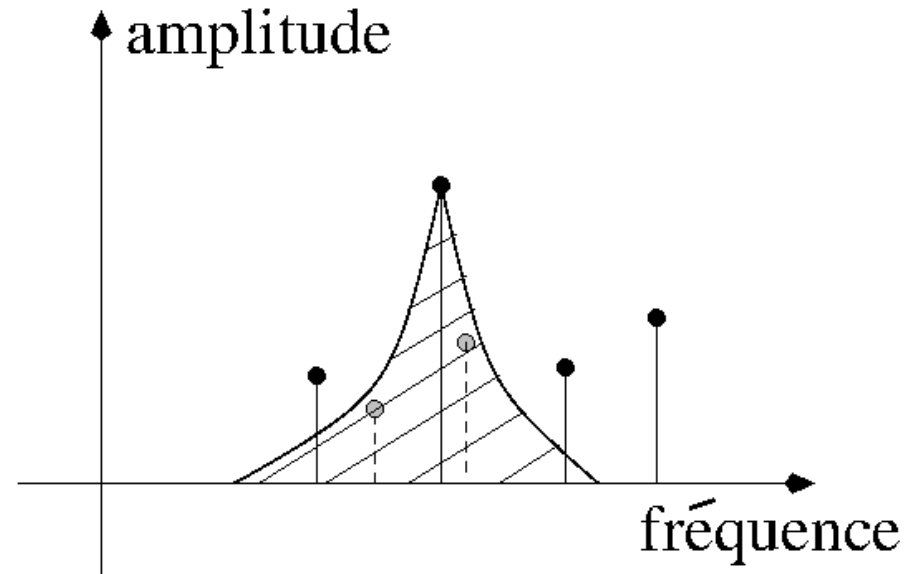
[wikipedia indexed colors]

# Compression avec pertes : son

- Et pour le son, comment procéder ?
  - Pour rappel, le son enregistré sur un CD est échantillonné à
    - 44.1 kHz, et chaque échantillon est encodé sur  $2 \times 16$  bits (2 canaux pour la stéréo).
  - Pour une seconde, il faut donc  $44100 \times 16 \times 2$  soit environ 1.4 Mégabits.
  - Le format MP3 permet d'encoder cette information sur 128 kilobits seulement, ce qui correspond-on à une réduction de 90% de la taille d'un fichier ! (sans déformation *sensiblement audible* du son)
- Quel est le « truc » ?

## Compression avec pertes : son

C'est (en partie) grâce à *l'effet de masque* : lorsqu'une sinusoïde avec une certaine fréquence est présente avec grande amplitude dans un son, elle cache à l'oreille humaine les autres sinusoïdes de fréquences proches et de moindre amplitude (c'est un effet *psychoacoustique*).



En conséquence, il n'y a pas besoin d'encoder une partie du signal avec autant de précision car on ne l'entend pas ou en tout cas beaucoup moins !

La même chose est vraie pour des sons rapprochés dans le temps.

## Compression avec pertes : son

Voici un lien vers une chanson a cappella de Suzanne Véga qui a posé des problèmes à Karlheinz Brandenburg et ses collègues dans la mise au point du format MP3 (autour de 1993)

<http://www.youtube.com/watch?v=mNWyF3iSMzs>

La méthode de compression avec perte qui fonctionnaient bien pour la plupart des morceaux de musique posait des problèmes pour ce type de chant *a cappella*.

C'est devenu un des morceaux de référence pour évaluer le MP3.



# Conclusion

Dans ce module, nous avons vu :

Un aperçu de la théorie de *l'échantillonnage des signaux*.

Un aperçu de la théorie de la *compression de données*.

Ces deux théories sont deux facettes d'un seul et même sujet...

...avec deux résultats centraux de même nature !

Si vous ne deviez retenir qu'une seule chose de ce module :

Il est toujours possible de stocker un signal ou des données en économisant de l'espace-mémoire

Il y a cependant des seuils à ne pas dépasser si on ne veut pas faire de pertes...

Les mesures de ces seuils sont respectivement la **bande passante** et **l'entropie**