# Compression de l'information

## I La compression en général

On peut considérer qu'un document, une image, une musique, une vidéo, un programme, ou un ensemble composé de ces éléments est une *information* au sens large du terme. Nous avons vu comment divers formats de données sont représentés en machine. Pour stocker ces informations ou pour les transmettre via Internet, on a tout intérêt à les *compresser*.

Compresser une information représentée par une séquence de bits $A$, c'est appliquer un algorithme qui va la transformer en une séquence B plus courte que A, mais dans laquelle l'information contenue dans A subsistera, soit à l'identique, soit de manière très similaire.
On range les algorithmes de compression en 2 catégories :

- La **compression de données sans perte :** l'algorithme de compression est une fonction $c$ (au sens mathématique du terme) qui transforme une séquence de bits $A$ en $c(A)$, telle que la longueur de $c(A)$ est inférieure à celle de $A$.
  La décompression est le processus inverse et est alors sans perte : on dispose d'une fonction $d$ avec la propriété suivante : quelle que soit la séquence de bits $A$, $d(c(A)) = A$.
  Autrement dit, en décompressant une séquence compressée, on obtient la séquence originale *à l'identique*.

- La **compression de données avec perte :** ne s'applique qu'aux données « perceptibles » telles que le son, l'image ou la vidéo. Les données à compresser peuvent parfois subir d'importantes modifications, et la perte d'information est irréversible, mais cela n'est pas ou peu perceptible par l'humain, et en tout cas, l'information abstraite contenue dans les données perceptibles reste transmise (par exemple, un fichier MP3 de très mauvaise qualité contenant un discours politique sera compris dans son intégralité en dépit de cette très mauvaise qualité).

**Définition**

On considère un algorithme de compression qui transforme une séquence de bits $A$ (non vide) en $B$.
Le *taux de compression* se calcule ainsi : $1 - \dfrac{\text{longueur de } B}{\text{longueur de } A}$

**Exercice 1**

Un algorithme d'encodage au format MP3 procède ainsi : il encode un morceau de musique de 5 minutes « qualité CD », c'est à dire stéréo, 16 bits, 44 100Hz, et produit un fichier de 5 Mo.
Quel est le taux de compression ?

# II Un exemple d'algorithme de compression : la compression JPEG

JPEG (Joint Photographic Experts Group) est un comité d'experts qui édite des normes de compression pour les images fixes. C'est aussi un format de fichier d'image. Le format JPEG standardisé a été adopté en 1992. Il est très populaire sur Internet, on le trouve aussi dans les appareils photo et les téléphones portables.

**Exercice 2**

- Ouvrir le fichier `tricoloring.png` avec GIMP (clic droit, ouvrir avec GIMP)
- Exporter ce fichier en format .jpg (CTRL+E). Activer l'aperçu dans une fenêtre d'image, faire varier le curseur de qualité et observer l'aperçu. Zoomer si nécessaire (CTRL + molette souris).
  Qu'observe-t-on ? À quelle catégorie l'algorithme de compression JPEG appartient-il ?
- Régler la qualité sur 70%, puis sauvegarder le résultat.
  Comparer les tailles des fichiers (dans l'explorateur Windows, faire : clic droit > propriétés) et calculer le taux de compression à $10^{-2}$ près.

# III Un algorithme de compression sans perte simple : RLE

Le *run-length encoding*, appelé en français le codage par plages, est sans doute de l'algorithme le plus simple auquel on puisse penser. Voici un exemple avec des mots :

- `aaaaaaaabbbbbbbbbbbbccccccccccc` devient `a8b12c10` (ici on y gagne).
- `abcdef` devient `a1b1c1d1e1f1` (ici on y perd).

**Exercice 3**

Pour une image, l'encodage RLE consiste à indiquer pour chaque suite de pixels d'une même couleur le nombre de pixels de cette séquence.

Vous allez travailler avec le fichier `rle.py`. Celui-ci utilise le module `image`, que nous avons déjà utilisé par le passé.

Coder la fonction `compresse` qui compresse une image suivant ce modèle :

- La fonction crée une liste. Les deux premiers éléments sont la largeur et la hauteur de l'image.
- Les éléments suivants vont 2 par deux : une liste de 3 éléments correspondant à une couleur de pixel, puis le nombre de fois que ce pixel se répète. Exemple : `[255,0,128],36`
- Tester la fonction sur le fichier `gamut.ppm`.
  Quelle est la taille, en `int` de l'image de départ ? Quelle est la taille, en `int` de la liste correspondant à l'image compressée ? Calculer le taux de compression.
- Recommencer avec le fichier `doudouchat.ppm`.
  Comment expliquer de tellles différences de taux ?

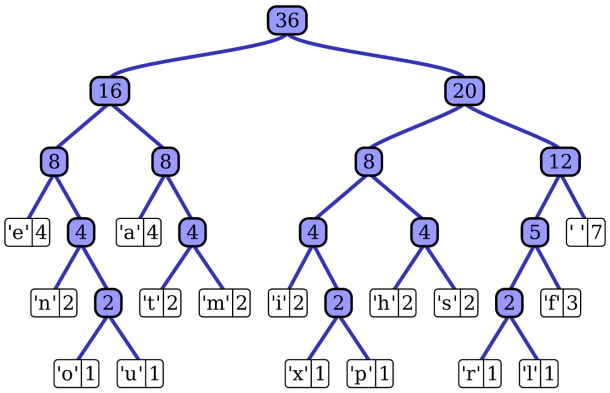
# IV Un algorithme performant : le codage de Huffman

C'est un algorithme de compression sans perte. Il est utilisé lors de la compression JPEG, en conjonction avec d'autres méthodes qui, elles, entrainent des pertes d'information. Il a inspiré les algorithmes de compression actuels, tel gzip, le logiciel libre de compression le plus utilisé actuellement.

Compressons le texte « this is an example of a huffman tree » : on commence par compter les occurences de chaque lettre du texte :

| o | u | x | p | r | l | n | t | m | i | h | s | f | e | a | espace |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 3 | 4 | 4 | 7 |

Puis on va créer un arbre dont les feuilles seront les lettres du texte.
Dans chaque nœud on stockera un poids et éventuellement une lettre.
On commence par créer les feuilles, puis on regroupe deux par deux les nœuds de poids minimal, en un nœud dont le poids est égal à la somme de ces deux poids.
On répète le processus jusqu'à ce qu'il ne reste plus qu'un nœud.
Pour finir, on choisit arbitrairement d'affecter 0 à « droite » et 1 à « gauche » (ou bien le contraire).




L'arbre que l'on obtient n'est pas unique car parfois on opère un choix parmi plusieurs regroupements valides.

Voici dans les grandes lignes comment l'arbre ci-dessus a été crée : On crée une variable `poids` qui prendra tout au long de l'algorithme le minimum des poids des feuilles , des nœuds construits (parmi les feuilles se trouvent les lettres non encore placées dans l'arbre).

- Le poids minimal est 1 : `poids=1`.

- On met o et u ensemble : poids 2, de même pour x et p et r et l (on aurait très bien pu mettre o avec x et p avec u).

- On a epuisé les feuilles de poids 1, `poids=2` donc on peut regrouper des nœuds de poids 2, ou bien ajouter des feuilles de poids 2 entre elles, ou bien mélanger.

- On arrive à 'f', `poids=3`. On le met avec un nœud ou une lettre de poids 2, on obtient alors un nœud de poids 5, mais attention : `poids=4`.

- Puisque `poids=4`, on regroupe les nœuds et les lettres de poids 4, puis `poids=5` et on n'a plus le choix : le nœud de poids 5 va avec l'espace (pour créer un nœud de poids 12), `poids=8`, on assemble, `poids=12` etc.

Ainsi la lettre 'u' (qui n'apparaît qu'une fois) est codée 11000. L'espace est, elle, codée 000. Plus un caractère est fréquent, plus il figure haut dans l'arbre, et plus son codage est court.

**Exercice 4**

Quelle est, en bits, la taille de « this is an example of a huffman tree » une fois le codage effectué ?

Puisque dans cette phrase, tous les caractères sont dans la table ANSI (7 bits par caractère), quel est le taux de compression ?

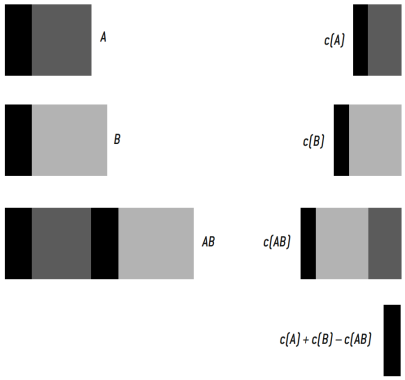
**Exercice 5 : Projet Huffman**

Créer un programme qui compresse des fichiers texte à l'aide de la méthode de Huffman.

# V Programmation - distance de similarité

## 1 En théorie

Compresser une information, cela revient *grosso modo* à éliminer les redondances qu'elle comporte. En partant de cette idée, on considère deux informations A et B représentées par au moins un bit (non nulles). Notons c un algorithme de compression performant (gzip par exemple) et AB le concaténé de A et de B (c'est-à-dire A et B « mis à la suite »). Supposons que A et B aient un contenu commun (en noir) alors lorsqu'on compresse AB, ce contenu commun n'est compté qu'une fois, de sorte que




$$c(A) + c(B) - c(AB)$$

est une bonne estimation de ce contenu commun.

On définit la **distance de similarité entre A et B** par

$$d(A,B) = 1 - \frac{c(A) + c(B) - c(AB)}{\max\left(c(A), c(B)\right)}$$

En théorie, ce que l'on vient de définir est une **distance** : quelles que soient les informations « non nulles » A, B, et C on a :

- $d(A,B) \geq 0$
- $d(A,B) = 0$ si et seulement si $A = B$.
- $d(A,B) = d(B,A)$ (symétrie)

– $d(A,C) \leqslant d(A,B) + d(B,C)$ (inégalité triangulaire)

## 2 Intérêts

On peut se servir de la distance de similarité pour classer des informations de même nature suivant leur ressemblance.

On a évalué ici les distances séparant 19 romans russes suivant la méthode précédente, puis on les a triés en rassemblant les plus proches. On a obtenu le diagramme ci-contre. Seul un roman de Tolstoï est mal classé.


Cette méthode est aussi appliquée en génétique : c'est l'ADN des différentes espèces que l'on compare, puis on établit un classement des espèces.

### Exercice 6

Nous allons étudier les distances séparant quelques langues à partir de fichiers texte : il s'agit de la Déclaration Universelle des Droits de l'Homme, traduite en diverses langues.
Vous allez travailler sur le fichier `classement_langues.py`.
Dans celui-ci, est déjà présente la fonction `taille_compressee` dont vous vous servirez.

**1.** Commencer par coder la fonction `distance`, qui donne la distance entre 2 fichiers.

**2.** Que vaut `distance('anglais','anglais')` ?
Que peut-on en déduire par rapport à la théorie ?

**3.** Comparer `distance('anglais','francais')` et `distance('francais','anglais')`.
Proposer une modification de la fonction `distance` pour qu'elle soit symétrique.

**4.** Coder la fonction `calcule_distances` qui, à partir d'une liste de fichiers, calcule les distances les séparant les uns des autres (attention : ne pas faire de calculs inutiles).

**5.** Exécuter

```
calcule_distances(['francais','portuguais','espagnol','tcheque',slovaque'])
```

Commenter les résultats.