C'est quoi le compactage, la compression ? Comment ça marche ?

En informatique, la place coûte cher. Les barettes mémoire et les disques durs coûtent de l'argent, alors on essai de ne pas trop les encombrer.

Si on prend 30 lettres A:

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Il faut **30 octets** dans la mémoire de l'ordinateur.

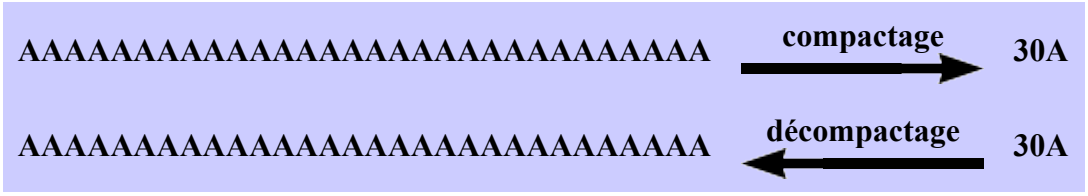
On pourrait la noter de façon plus concise : "*30 fois la lettre A*":

30A

Là, ça n'occupe plus que **3 octets** !

C'est ça la compression (ou compactage) ! Pour une même donnée, on essai de trouver la représentation la plus compacte possible.

On peut facilement passer de l'un à l'autre:



Il existe des tas de méthodes différentes et astucieuses pour gagner de la place !

En voici quelques unes.

RLE (Run Length Encoding)

C'est exactement notre exemple ci-dessus. On repère les répétitions et on les note de façon plus compacte.

On peut par exemple utiliser un symbole spécial, comme * pour indiquer une répétition.

C'est à dire:

***5A**	on répète 5 fois A
BBB	on laisse tel quel (*3B n'est pas plus compacte que BBB)
C	on laisse tel quel
***4D**	on répète 4 fois D (c'est plus compacte que DDDD)

Ce système de compactage est utilisé par exemple dans le format PCX et dans certaines variantes du

format TIFF.

Huffman

Huffman est un système de codage statistique.

Pour Huffman, il faut compter la fréquence de chaque caractère.

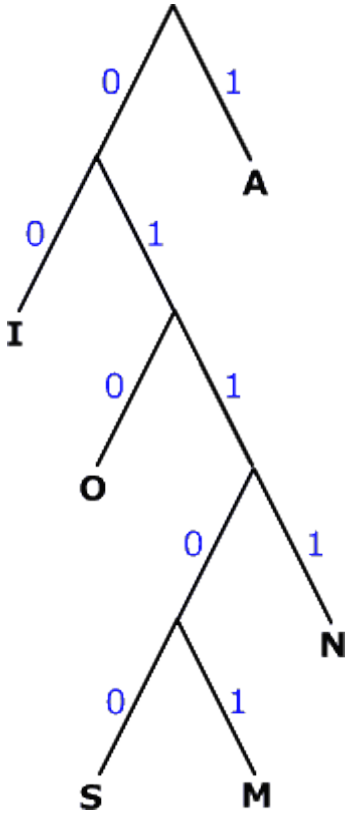
Imaginons que nous ayons compté la fréquence des lettres dans un texte.

A	I	M	N	O	S
92	30	11	17	28	3

(dans notre texte, on avait 92 lettres A, 30 lettres I, etc.)

La plus petite unité d'information que peut manipuler l'ordinateur est le bit : il vaut 0 ou 1.

On peut construire un arbre binaire:



Plus une lettre est fréquente, moins on utilise de bits pour la représenter.

La lettre **A** (la plus fréquente) est représentée par un seul bit : **1**.
Pour **I** (un peu moins fréquente), on utilise deux bits : **00**.
Pour **O** (encore moins fréquente), on utilise 3 bits : **010**.
Et ainsi de suite.

En binaire, le mot '**MAISON**' s'écrit normalement (en utilisant le code ASCII):

M	A	I	S	O	N
1001101	**1000001**	**1001001**	**1010011**	**1001111**	**1001110**

En Huffman (avec notre arbre ci-dessus), il devient:

M	A	I	S	O	N
01101	1	00	01100	010	0111

On a clairement gagné de la place !
Avant compression, notre texte prenait 48 bits (6 octets).
Après compression, il ne prend plus que 20 bits (2,5 octets).

Mais Huffman a un inconvénient : il faut faire une analyse statistique de ***tout*** le texte avant de pouvoir compacter. Cela peut prendre du temps.

Huffman est assez peu utilisé, surtout face aux performances d'autres algorithmes comme Lempel-Ziv.

Lempel-Ziv

Lempel-Ziv, lui, se construit au fur et à mesure un dictionnaire numéroté des 'mots' déjà rencontrés. S'il tombe sur un mot qu'il a déjà rencontré, il ne le recopie pas mais il note seulement son 'numéro'.

Voici l'algorithme de compactage:

```
w = (vide)
Boucle
    lire le caractère suivant dans K
    Si w+K est dans le dictionnaire:
        w = w+K
    sinon
        sortir le code de w
        ajouter w+K au dictionnaire
        w = K
    fin_si
fin_boucle
```

On commence avec un dictionnaire rempli avec tous les caractères existants (0 à 255). (Par exemple, 65 contient **A**, 66 contient **B**, etc.).
Les nouveaux mots ajoutés au dictionnaire commencent donc au numéro 256.

Exemple:

La chaîne **/WED/WE/WEE/WEB**.
Il faut 15*8 = **120 bits** pour stocker cette chaîne en mémoire.

Compactons-la avec Lempel-Ziv:

Caractères lu	Code sorti	Ajout au dictionnaire
/		/ est dans le dictionnaire.
W	47 (code ASCII de /)	256 = /W
E	87 (code ASCII de W)	257 = WE
D	69 (code ASCII de E)	258 = ED
/	68 (code ASCII de D)	259 = D/
W		/W est dans le dictionnaire.

E	256 (code de /W)	260 = /WE
/	69 (code ASCII de E)	261 = E/
W		/W est dans le dictionnaire.
E		/WE est dans le dictionnaire.
E	260 (code de /WE)	262 = /WEE
/		E/ est dans le dictionnaire.
W	261 (code de E/)	263 = E/W
E		WE est dans le dictionnaire.
B	257 (code de WE)	264 = WEB
(fin)	66 (code ASCII de B)	

L'algorithme sort : '**/WED**<256>**E**<260><261><257>**B**'.

Ici, il ne faut plus que 4*8 + 6*9 = **86 bits**. (après **/WED**, on dépasse 255 : il faut utiliser 9 bits).

Des variantes de cette méthode de compression sont utilisées dans les format ZIP, RAR, ARJ, CAB, GIF, PNG, TIFF...

C'est une méthode très populaire car:

- elle est performante : elle compacte bien toutes sortes de données.
- elle n'est pas très difficile à programmer
- elle peut travailler sur des données comme elles arrivent : pas besoin d'analyser toutes les données à l'avance comme avec Huffman.
- on peut utiliser des dictionnaires plus ou moins gros en fonction de la mémoire que l'on a. (Avec de petits dictionnaires, on compresse un peu moins bien, mais on compresse plus vite).

Il existe de nombreuses variantes de LZ: LZ77, LZW, etc.

D'autres méthodes

Il existe d'autres méthodes comme le codage arithmétique, le codage par dictionnaire fixe (comme les Fax) ou l'algorithme Burrows-Wheeler. Certaines ont de bien meilleures performances que LZ, ZIP et Huffman en taux de compression.

Les méthodes que nous avons vu ici dont dites **non-destructives**, car on peut décompresser les données telles qu'elles étaient avant compression.

Il existe des méthodes **destructives** qui détruisent une partie des données à la compression. En décompressant, on a plus exactement les données d'origine, mais des données très proches. C'est le cas pour le MP3 ou le JPEG.