

# Théorie de l'Information

## Algorithmes de compression

Julien Lavauzelle

Université Paris 8

Master 1 ACC – Théorie de l'Information

20/11/2023

1. Algorithme de Lempel–Ziv

2. Compression de données

1. Algorithme de Lempel–Ziv

2. Compression de données

**Définition.** Soit  $\mathcal{C} = (C_n)_{n \geq 1}$  une famille de codes telles que pour tout  $n \geq 1$ , le code  $C_n$  encode des séquences de  $n$  symboles. Alors, la famille  $\mathcal{C}$  est dite **universelle** si

$$\lim_{n \rightarrow \infty} \mathbb{E}_X \left[ \frac{1}{n} \ell(C_n(X_1, \dots, X_n)) \right] = \bar{H}(X)$$

pour toutes les sources stationnaires  $X$ .

**En termes informels :**  $\mathcal{C}$  est une famille de code universels si, asymptotiquement, sa longueur moyenne par symbole est égale à l'entropie de la source.

C'est le mieux qu'on puisse espérer.

**Objectif :** construire un code qui soit (presque) universel

**Objectif :** construire un code qui soit (presque) universel

Il existe les algorithmes **LZ77**, **LZ78** de Lempel et Ziv, utilisés comme brique de base dans beaucoup de **logiciels de compression actuels**.

**Objectif :** construire un code qui soit (presque) universel

Il existe les algorithmes **LZ77**, **LZ78** de Lempel et Ziv, utilisés comme brique de base dans beaucoup de **logiciels de compression actuels**.

**Idée.** Étant donné une chaîne de caractère en entrée :

1. On parcourt la chaîne de caractères, que l'on découpe en suite de mots distincts de taille minimale.
2. Lors de ce parcours, on tient à jour un dictionnaire des mots trouvés, et une liste ordonnée de références vers ce dictionnaire (pointeurs).

**Objectif :** construire un code qui soit (presque) universel

Il existe les algorithmes **LZ77**, **LZ78** de Lempel et Ziv, utilisés comme brique de base dans beaucoup de **logiciels de compression actuels**.

**Idée.** Étant donné une chaîne de caractère en entrée :

1. On parcourt la chaîne de caractères, que l'on découpe en suite de mots distincts de taille minimale.
2. Lors de ce parcours, on tient à jour un dictionnaire des mots trouvés, et une liste ordonnée de références vers ce dictionnaire (pointeurs).

**Exemple typique.** Sur l'alphabet  $\{a, b\}$ , on considère la séquence suivante :

*aababbabbaabbababbabaabbabaaaabbabaaa*



**Objectif :** construire un code qui soit (presque) universel

Il existe les algorithmes **LZ77**, **LZ78** de Lempel et Ziv, utilisés comme brique de base dans beaucoup de **logiciels de compression actuels**.

**Idée.** Étant donné une chaîne de caractère en entrée :

1. On parcourt la chaîne de caractères, que l'on découpe en suite de mots distincts de taille minimale.
2. Lors de ce parcours, on tient à jour un dictionnaire des mots trouvés, et une liste ordonnée de références vers ce dictionnaire (pointeurs).

**Exemple typique.** Sur l'alphabet  $\{a, b\}$ , on considère la séquence suivante :

*aababbabbaabbababbabaabbabaaaabbabaaa*

On peut la découper en :

*a|ab|abb|abba|abbab|abbaba|abbabaa|abbabaaa*

**Objectif :** construire un code qui soit (presque) universel

Il existe les algorithmes **LZ77**, **LZ78** de Lempel et Ziv, utilisés comme brique de base dans beaucoup de **logiciels de compression actuels**.

**Idée.** Étant donné une chaîne de caractère en entrée :

1. On parcourt la chaîne de caractères, que l'on découpe en suite de mots distincts de taille minimale.
2. Lors de ce parcours, on tient à jour un dictionnaire des mots trouvés, et une liste ordonnée de références vers ce dictionnaire (pointeurs).

**Exemple typique.** Sur l'alphabet  $\{a, b\}$ , on considère la séquence suivante :

*aababbabbaabbababbabaabbabaaaabbabaaa*

On peut la découper en :

*a|ab|abb|abba|abbab|abbaba|abbabaa|abbabaaa*

Puis on la transforme en :

*0a|1b|2b|3a|4b|5a|6a|7a*

où le nombre  $i$  fait référence au  $i$ -ème mot de la liste.

On présente ici l'algorithme LZW (pour Lempel-Ziv-Welch), une variante de LZ78.

## ALGORITHME DE LEMPEL-ZIV-WELCH (LZW)

**Entrée :** une chaîne de caractères  $x \in \mathcal{X}^m$

**Sortie :** une séquences de couples  $((i_j, a_j))_{j \geq 1}$   
où  $i_j \in \mathbb{N}$  et  $a_j \in \mathcal{X}$

1. Initialiser :  $\text{res} \leftarrow []$ ,  $\text{dict} \leftarrow [""]$ ,  $\text{buffer} \leftarrow ""$ ,  $i = 0$
2. **Pour**  $j$  allant de 1 à  $m$  :
  - 2.1 Ajouter  $x_j$  à la fin de  $\text{buffer}$
  - 2.2 **Si** : il existe  $k$  tel que  $\text{buffer} = \text{dict}[k]$ , **alors** :  
 $i = k$
  - 2.3 **Sinon** :  
Ajouter  $\text{buffer}$  à  $\text{dict}$   
Ajouter  $(i, x_j)$  à  $\text{res}$   
Réinitialiser  $\text{buffer}$  à ""  
Réinitialiser  $i$  à 0
3. **Retourner** :  $\text{res}$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont : `res = []`  
`dict = []`  
`buffer =`

**Illustration** de l'état du découpage :

$$x = |bbaabbabaaaaababbbbbaabbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont : `res = []`

`dict = []`

`buffer = b`

**Illustration** de l'état du découpage :

$$x = |b|baabbabaaaaababbbbbaabbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $res = [(0, b)]$

$dict = [b]$

$buffer = b$

**Illustration** de l'état du découpage :

$$x = |b|b|aabbabaaaaababbbbbaabbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b)]$

$\text{dict} = [b]$

$\text{buffer} = ba$

**Illustration** de l'état du découpage :

$$x = |b|ba|abbabaaaaababbbbbaabbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a)]$

$\text{dict} = [b, ba]$

$\text{buffer} = a$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bbabaaaaababbbbbaabbabb$$



## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a)]$

$\text{dict} = [b, ba, a]$

$\text{buffer} = b$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|b|babaaaaababbbbbaabbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a)]$

$\text{dict} = [b, ba, a]$

$\text{buffer} = bb$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|abaaaaababbbbbaabbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b)]$

$\text{dict} = [b, ba, a, bb]$

$\text{buffer} = a$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|a|baaaaababbbbbaaabbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b)]$

$\text{dict} = [b, ba, a, bb]$

$\text{buffer} = ab$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aaaaababbbbbaabbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b)]$

$\text{dict} = [b, ba, a, bb, ab]$

$\text{buffer} = a$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|a|aaaaababbbbbaabbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b)]$

$\text{dict} = [b, ba, a, bb, ab]$

$\text{buffer} = aa$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aa|aaababbbbbaabbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b), (3, a)]$

$\text{dict} = [b, ba, a, bb, ab, aa]$

$\text{buffer} = a$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aa|a|aababbbbbaabbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b), (3, a)]$

$\text{dict} = [b, ba, a, bb, ab, aa]$

$\text{buffer} = aa$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aa|aa|ababbbbbaabbabb$$



## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b), (3, a)]$

$\text{dict} = [b, ba, a, bb, ab, aa]$

$\text{buffer} = aaa$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aa|aaa|babbbbbaaabbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b), (3, a), (6, a)]$

$\text{dict} = [b, ba, a, bb, ab, aa, aaa]$

$\text{buffer} = b$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aa|aaa|b|abbbbbaabbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b), (3, a), (6, a)]$

$\text{dict} = [b, ba, a, bb, ab, aa, aaa]$

$\text{buffer} = ba$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aa|aaa|ba|bbbbaabbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b), (3, a), (6, a)]$

$\text{dict} = [b, ba, a, bb, ab, aa, aaa]$

$\text{buffer} = bab$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aa|aaa|bab|bbbaabbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b), (3, a), (6, a), (2, b)]$   
 $\text{dict} = [b, ba, a, bb, ab, aa, aaa, bab]$   
 $\text{buffer} = b$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aa|aaa|bab|b|bbaabbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b), (3, a), (6, a), (2, b)]$   
 $\text{dict} = [b, ba, a, bb, ab, aa, aaa, bab]$   
 $\text{buffer} = bb$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aa|aaa|bab|bb|baabbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b), (3, a), (6, a), (2, b)]$   
 $\text{dict} = [b, ba, a, bb, ab, aa, aaa, bab]$   
 $\text{buffer} = bb$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aa|aaa|bab|bbb|aaabbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b), (3, a), (6, a), (2, b), (4, b)]$   
 $\text{dict} = [b, ba, a, bb, ab, aa, aaa, bab, bbb]$   
 $\text{buffer} = a$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aa|aaa|bab|bbb|a|aabbabb$$



## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b), (3, a), (6, a), (2, b), (4, b)]$   
 $\text{dict} = [b, ba, a, bb, ab, aa, aaa, bab, bbb]$   
 $\text{buffer} = aa$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aa|aaa|bab|bbb|aa|abbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b), (3, a), (6, a), (2, b), (4, b)]$   
 $\text{dict} = [b, ba, a, bb, ab, aa, aaa, bab, bbb]$   
 $\text{buffer} = aaa$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aa|aaa|bab|bbb|aaa|bbabb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b), (3, a), (6, a), (2, b), (4, b)]$   
 $\text{dict} = [b, ba, a, bb, ab, aa, aaa, bab, bbb]$   
 $\text{buffer} = aaab$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aa|aaa|bab|bbb|aaab|babb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b), (3, a), (6, a), (2, b), (4, b), (7, a)]$   
 $\text{dict} = [b, ba, a, bb, ab, aa, aaa, bab, bbb, aaab]$   
 $\text{buffer} = b$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aa|aaa|bab|bbb|aaab|b|abb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b), (3, a), (6, a), (2, b), (4, b), (7, a)]$   
 $\text{dict} = [b, ba, a, bb, ab, aa, aaa, bab, bbb, aaab]$   
 $\text{buffer} = ba$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aa|aaa|bab|bbb|aaab|ba|bb$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b), (3, a), (6, a), (2, b), (4, b), (7, a)]$   
 $\text{dict} = [b, ba, a, bb, ab, aa, aaa, bab, bbb, aaab]$   
 $\text{buffer} = bab$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aa|aaa|bab|bbb|aaab|bab|b$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b), (3, a), (6, a), (2, b), (4, b), (7, a)]$   
 $\text{dict} = [b, ba, a, bb, ab, aa, aaa, bab, bbb, aaab]$   
 $\text{buffer} = babb$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aa|aaa|bab|bbb|aaab|babb|$$

## Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne de caractères en entrée est :

$$x = bbaabbabaaaaababbbbbaabbabb$$

Les variables de calcul sont :  $\text{res} = [(0, b), (1, a), (0, a), (1, b), (3, b), (3, a), (6, a), (2, b), (4, b), (7, a), (8, b)]$   
 $\text{dict} = [b, ba, a, bb, ab, aa, aaa, bab, bbb, aaab, babb]$   
 $\text{buffer} =$

**Illustration** de l'état du découpage :

$$x = |b|ba|a|bb|ab|aa|aaa|bab|bbb|aaab|babb|$$



**Question :**

Ce codage est-il « bon » ?

Question :

Ce codage est-il « bon » ?

Deux analyses :

- **en pire cas** : existe-t-il des chaînes de caractères dont la taille augmente grossièrement ?
- **en cas moyen** : pour un processus stochastique donné, quelle est la longueur moyenne du codage LZW ?

**Objectif :** trouver une chaîne de caractères  $x$  pour laquelle le codage LZW est (asymptotiquement) le plus long.

**Objectif** : trouver une chaîne de caractères  $x$  pour laquelle le codage LZW est (asymptotiquement) le plus long.

Lors du découpage, pour avoir un codage **long**, faut-il que  $x$  soit divisée en beaucoup de petits mots ? ou en peu de mots ressemblants et longs ?

**Objectif** : trouver une chaîne de caractères  $x$  pour laquelle le codage LZW est (asymptotiquement) le plus long.

Lors du découpage, pour avoir un codage **long**, faut-il que  $x$  soit divisée en beaucoup de petits mots ? ou en peu de mots ressemblants et longs ?

**Rappel** : l'exemple typique donne un codage assez court

$$a|ab|abb|abba|abbab|abbaba|abbabaa|abbabaaa \mapsto 0a|1b|2b|3a|4b|5a|6a|7a$$

car le  $n$ -ième mot, de longueur  $n$ , est codé avec  $\simeq \log_2 n$  bits.

**Objectif** : trouver une chaîne de caractères  $x$  pour laquelle le codage LZW est (asymptotiquement) le plus long.

Lors du découpage, pour avoir un codage **long**, faut-il que  $x$  soit divisée en beaucoup de petits mots ? ou en peu de mots ressemblants et longs ?

**Rappel** : l'exemple typique donne un codage assez court

$$a|ab|abb|abba|abbab|abbaba|abbabaa|abbabaaa \mapsto 0a|1b|2b|3a|4b|5a|6a|7a$$

car le  $n$ -ième mot, de longueur  $n$ , est codé avec  $\simeq \log_2 n$  bits.

Pour avoir un **cas défavorable**, il faut faire le contraire. Pour  $k \geq 1$ , la chaîne

$$x^{(k)} = a|b|aa|ab|ba|bb|aaa| \cdots \cdots | \underbrace{bbb \dots bb}_{k \text{ fois}}$$

donne le mot

$$c^{(k)} = 0a | 0b | 1a | 1b | 2a | 2b | 3a | \cdots \cdots | ((k-2)2^k - 2)b$$

**Objectif** : trouver une chaîne de caractères  $x$  pour laquelle le codage LZW est (asymptotiquement) le plus long.

Lors du découpage, pour avoir un codage **long**, faut-il que  $x$  soit divisée en beaucoup de petits mots ? ou en peu de mots ressemblants et longs ?

**Rappel** : l'exemple typique donne un codage assez court

$$a|ab|abb|abba|abbab|abbaba|abbabaa|abbabaaa \mapsto 0a|1b|2b|3a|4b|5a|6a|7a$$

car le  $n$ -ième mot, de longueur  $n$ , est codé avec  $\simeq \log_2 n$  bits.

Pour avoir un **cas défavorable**, il faut faire le contraire. Pour  $k \geq 1$ , la chaîne

$$x^{(k)} = a|b|aa|ab|ba|bb|aaa| \cdots \cdots | \underbrace{bbb \dots bb}_{k \text{ fois}}$$

donne le mot

$$c^{(k)} = 0a | 0b | 1a | 1b | 2a | 2b | 3a | \cdots \cdots | ((k-2)2^k - 2)b$$

**Question** : quelle est la longueur de  $c^{(k)}$  par rapport à celle de  $x^{(k)}$  ?

Soit  $x^{(k)} = a|b|aa|ab|ba|bb|aaa| \dots \dots | \underbrace{bbb \dots bb}_{k \text{ fois}}$  le mot à coder.



Soit  $\mathbf{x}^{(k)} = a|b|aa|ab|ba|bb|aaa| \dots \dots | \underbrace{bbb \dots bb}_{k \text{ fois}}$  le mot à coder.

On montre que  $\ell(\mathbf{x}^{(k)}) = \sum_{i=1}^k i2^i = (k-1)2^{k+1} + 2$

— **preuve** : exercice.

Soit  $\mathbf{x}^{(k)} = a|b|aa|ab|ba|bb|aaa| \dots \dots | \underbrace{bbb \dots bb}_{k \text{ fois}}$  le mot à coder.

On montre que  $\ell(\mathbf{x}^{(k)}) = \sum_{i=1}^k i2^i = (k-1)2^{k+1} + 2$

— **preuve** : exercice.

On note  $w(\mathbf{x}^{(k)})$  le nombre de mots dans la division de la chaîne  $\mathbf{x}^{(k)}$ . On a :

$$w(\mathbf{x}^{(k)}) = \sum_{i=1}^k 2^i = 2^{k+1} - 2.$$

Soit  $\mathbf{x}^{(k)} = a|b|aa|ab|ba|bb|aaa| \dots \dots | \underbrace{bbb \dots bb}_{k \text{ fois}}$  le mot à coder.

On montre que  $\ell(\mathbf{x}^{(k)}) = \sum_{i=1}^k i2^i = (k-1)2^{k+1} + 2$

— **preuve** : exercice.

On note  $w(\mathbf{x}^{(k)})$  le nombre de mots dans la division de la chaîne  $\mathbf{x}^{(k)}$ . On a :

$$w(\mathbf{x}^{(k)}) = \sum_{i=1}^k 2^i = 2^{k+1} - 2.$$

Enfin, chaque indice de préfixe est codé sur  $\lceil \log w(\mathbf{x}^{(k)}) \rceil = k+1$  bits, donc

$$\ell(\mathbf{c}^{(k)}) = w(\mathbf{x}^{(k)}) \times (k+1) = \frac{k+1}{k-1} (\ell(\mathbf{x}^{(k)}) + 2) - 2.$$

Soit  $\mathbf{x}^{(k)} = a|b|aa|ab|ba|bb|aaa| \dots \dots | \underbrace{bbb \dots bb}_{k \text{ fois}}$  le mot à coder.

On montre que  $\ell(\mathbf{x}^{(k)}) = \sum_{i=1}^k i2^i = (k-1)2^{k+1} + 2$

— **preuve** : exercice.

On note  $w(\mathbf{x}^{(k)})$  le nombre de mots dans la division de la chaîne  $\mathbf{x}^{(k)}$ . On a :

$$w(\mathbf{x}^{(k)}) = \sum_{i=1}^k 2^i = 2^{k+1} - 2.$$

Enfin, chaque indice de préfixe est codé sur  $\lceil \log w(\mathbf{x}^{(k)}) \rceil = k+1$  bits, donc

$$\ell(\mathbf{c}^{(k)}) = w(\mathbf{x}^{(k)}) \times (k+1) = \frac{k+1}{k-1} (\ell(\mathbf{x}^{(k)}) + 2) - 2.$$

Le ratio qui nous intéresse est le rapport entre la longueur du mot encodé et celle du mot initial. Si l'on note  $n = \ell(\mathbf{x}^{(k)})$ , on a alors :

$$\frac{\ell(\mathbf{c}^{(k)})}{\ell(\mathbf{x}^{(k)})} = \left(1 + \frac{2}{k-1}\right) \left(1 + \frac{2}{\ell(\mathbf{x}^{(k)})}\right) - \frac{2}{\ell(\mathbf{x}^{(k)})} \stackrel{n \rightarrow \infty}{=} 1 + o\left(\frac{1}{\log_2 n}\right).$$

Soit  $\mathbf{x}^{(k)} = a|b|aa|ab|ba|bb|aaa| \dots \dots | \underbrace{bbb \dots bb}_{k \text{ fois}}$  le mot à coder.

On montre que  $\ell(\mathbf{x}^{(k)}) = \sum_{i=1}^k i2^i = (k-1)2^{k+1} + 2$

— **preuve** : exercice.

On note  $w(\mathbf{x}^{(k)})$  le nombre de mots dans la division de la chaîne  $\mathbf{x}^{(k)}$ . On a :

$$w(\mathbf{x}^{(k)}) = \sum_{i=1}^k 2^i = 2^{k+1} - 2.$$

Enfin, chaque indice de préfixe est codé sur  $\lceil \log w(\mathbf{x}^{(k)}) \rceil = k+1$  bits, donc

$$\ell(\mathbf{c}^{(k)}) = w(\mathbf{x}^{(k)}) \times (k+1) = \frac{k+1}{k-1} (\ell(\mathbf{x}^{(k)}) + 2) - 2.$$

Le ratio qui nous intéresse est le rapport entre la longueur du mot encodé et celle du mot initial. Si l'on note  $n = \ell(\mathbf{x}^{(k)})$ , on a alors :

$$\frac{\ell(\mathbf{c}^{(k)})}{\ell(\mathbf{x}^{(k)})} = \left(1 + \frac{2}{k-1}\right) \left(1 + \frac{2}{\ell(\mathbf{x}^{(k)})}\right) - \frac{2}{\ell(\mathbf{x}^{(k)})} \stackrel{n \rightarrow \infty}{=} 1 + o\left(\frac{1}{\log_2 n}\right).$$

Autrement dit, en pire cas l'algorithme LZW est asymptotiquement optimal.

Pour le cas moyen, on doit faire une hypothèse supplémentaire sur la source.

Processus **ergodique** (*définition informelle*) : processus stochastique dont les statistiques (par exemple, sa moyenne, son entropie) peuvent être approchées à partir d'une réalisation suffisamment longue.

Pour le cas moyen, on doit faire une hypothèse supplémentaire sur la source.

Processus **ergodique** (*définition informelle*) : processus stochastique dont les statistiques (par exemple, sa moyenne, son entropie) peuvent être approchées à partir d'une réalisation suffisamment longue.

**Proposition.** Soit  $C(\mathbf{x}^n)$  le codage LZW d'un message  $\mathbf{x}^n$  de longueur  $n$ , issu d'un processus stochastique  $\mathbf{X}$  stationnaire et ergodique. Alors on a

$$\lim_{n \rightarrow +\infty} \frac{1}{n} \ell(C(\mathbf{x}^n)) = \bar{H}(\mathbf{X}).$$

Pour le cas moyen, on doit faire une hypothèse supplémentaire sur la source.

Processus **ergodique** (*définition informelle*) : processus stochastique dont les statistiques (par exemple, sa moyenne, son entropie) peuvent être approchées à partir d'une réalisation suffisamment longue.

**Proposition.** Soit  $C(\mathbf{x}^n)$  le codage LZW d'un message  $\mathbf{x}^n$  de longueur  $n$ , issu d'un processus stochastique  $\mathbf{X}$  stationnaire et ergodique. Alors on a

$$\lim_{n \rightarrow +\infty} \frac{1}{n} \ell(C(\mathbf{x}^n)) = \bar{H}(\mathbf{X}).$$

L'hypothèse supplémentaire d'ergodicité de la source ne permet pas de dire que le codage de Lempel-Ziv-Welch est universel. On le qualifie néanmoins de **ponctuellement universel**.



1. Algorithme de Lempel–Ziv

2. Compression de données

Dans la vie courante, les fichiers et messages sont structurés.

Dans la vie courante, les fichiers et messages sont structurés.

**But** : réduire leur taille, pour les stocker/envoyer de manière efficace.

Dans la vie courante, les fichiers et messages sont structurés.

**But :** réduire leur taille, pour les stocker/envoyer de manière efficace.

**Exemple :** en français, certaines suites de lettres sont plus fréquentes que d'autres, même si elles contiennent les mêmes lettres.

ES	SE		LE	EL		DE	ED		QU	UQ
3,05%	1,32%		2,22%	1,42%		2,17%	1,01 %		1,11 %	0,02%

Dans la vie courante, les fichiers et messages sont structurés.

**But :** réduire leur taille, pour les stocker/envoyer de manière efficace.

**Exemple :** en français, certaines suites de lettres sont plus fréquentes que d'autres, même si elles contiennent les mêmes lettres.

ES	SE		LE	EL		DE	ED		QU	UQ
3,05%	1,32%		2,22%	1,42%		2,17%	1,01 %		1,11 %	0,02%

**Question :** comment utiliser cette dépendance à notre profit ?

$x = \text{aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbb}$

$x = \text{aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbb}$

$$C(x) = 29a7b$$













**Algorithme RLE** : *run-length encoding*, codage par plages

**Algorithme RLE** : *run-length encoding*, codage par plages

On découpe le texte en plages de caractères identiques. Chaque plage est ensuite encodée par un nombre suivi du caractère répété.

**Algorithme RLE** : *run-length encoding*, codage par plages

On découpe le texte en plages de caractères identiques. Chaque plage est ensuite encodée par un nombre suivi du caractère répété.

C'est le premier exemple :

$$x = \text{aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbb} \mapsto C(x) = 29a7b.$$





Pour un alphabet  $\mathcal{X}$ , on note  $s = \lceil \log_2 |\mathcal{X}| \rceil$ .

Pour un alphabet  $\mathcal{X}$ , on note  $s = \lceil \log_2 |\mathcal{X}| \rceil$ .

**Meilleur cas :** un message avec  $n$  fois la même lettre  $x \in \mathcal{X}$ .

Pour un alphabet  $\mathcal{X}$ , on note  $s = \lceil \log_2 |\mathcal{X}| \rceil$ .

**Meilleur cas :** un message avec  $n$  fois la même lettre  $x \in \mathcal{X}$ .

- ▶ Longueur (en bits) du message :

Pour un alphabet  $\mathcal{X}$ , on note  $s = \lceil \log_2 |\mathcal{X}| \rceil$ .

**Meilleur cas :** un message avec  $n$  fois la même lettre  $x \in \mathcal{X}$ .

- ▶ Longueur (en bits) du message :  $n \times s$

Pour un alphabet  $\mathcal{X}$ , on note  $s = \lceil \log_2 |\mathcal{X}| \rceil$ .

**Meilleur cas :** un message avec  $n$  fois la même lettre  $x \in \mathcal{X}$ .

- ▶ Longueur (en bits) du message :  $n \times s$
- ▶ Longueur (en bits) du message compressé :

Pour un alphabet  $\mathcal{X}$ , on note  $s = \lceil \log_2 |\mathcal{X}| \rceil$ .

**Meilleur cas :** un message avec  $n$  fois la même lettre  $x \in \mathcal{X}$ .

- ▶ Longueur (en bits) du message :  $n \times s$
- ▶ Longueur (en bits) du message compressé :  $\lceil \log_2 n \rceil + s$

Pour un alphabet  $\mathcal{X}$ , on note  $s = \lceil \log_2 |\mathcal{X}| \rceil$ .

**Meilleur cas :** un message avec  $n$  fois la même lettre  $x \in \mathcal{X}$ .

- ▶ Longueur (en bits) du message :  $n \times s$
- ▶ Longueur (en bits) du message compressé :  $\lceil \log_2 n \rceil + s$

**Pire cas :** un message avec une succession de  $n$  symboles différents.

Pour un alphabet  $\mathcal{X}$ , on note  $s = \lceil \log_2 |\mathcal{X}| \rceil$ .

**Meilleur cas :** un message avec  $n$  fois la même lettre  $x \in \mathcal{X}$ .

- ▶ Longueur (en bits) du message :  $n \times s$
- ▶ Longueur (en bits) du message compressé :  $\lceil \log_2 n \rceil + s$

**Pire cas :** un message avec une succession de  $n$  symboles différents.

- ▶ Longueur (en bits) du message :



Pour un alphabet  $\mathcal{X}$ , on note  $s = \lceil \log_2 |\mathcal{X}| \rceil$ .

**Meilleur cas :** un message avec  $n$  fois la même lettre  $x \in \mathcal{X}$ .

- ▶ Longueur (en bits) du message :  $n \times s$
- ▶ Longueur (en bits) du message compressé :  $\lceil \log_2 n \rceil + s$

**Pire cas :** un message avec une succession de  $n$  symboles différents.

- ▶ Longueur (en bits) du message :  $n \times s$

Pour un alphabet  $\mathcal{X}$ , on note  $s = \lceil \log_2 |\mathcal{X}| \rceil$ .

**Meilleur cas :** un message avec  $n$  fois la même lettre  $x \in \mathcal{X}$ .

- ▶ Longueur (en bits) du message :  $n \times s$
- ▶ Longueur (en bits) du message compressé :  $\lceil \log_2 n \rceil + s$

**Pire cas :** un message avec une succession de  $n$  symboles différents.

- ▶ Longueur (en bits) du message :  $n \times s$
- ▶ Longueur (en bits) du message compressé :

Pour un alphabet  $\mathcal{X}$ , on note  $s = \lceil \log_2 |\mathcal{X}| \rceil$ .

**Meilleur cas :** un message avec  $n$  fois la même lettre  $x \in \mathcal{X}$ .

- ▶ Longueur (en bits) du message :  $n \times s$
- ▶ Longueur (en bits) du message compressé :  $\lceil \log_2 n \rceil + s$

**Pire cas :** un message avec une succession de  $n$  symboles différents.

- ▶ Longueur (en bits) du message :  $n \times s$
- ▶ Longueur (en bits) du message compressé :  $n(1 + s)$

Pour des **symboles tirés uniformément et indépendamment**, la probabilité d'avoir  $n$  symboles identiques consécutifs décroît exponentiellement avec  $n$ .

Pour des **symboles tirés uniformément et indépendamment**, la probabilité d'avoir  $n$  symboles identiques consécutifs décroît exponentiellement avec  $n$ .

On peut montrer que pour un processus dont les symboles sont tirés uniformément et indépendamment, **en moyenne le codage RLE ne raccourcit pas le message.**

Pour des **symboles tirés uniformément et indépendamment**, la probabilité d'avoir  $n$  symboles identiques consécutifs décroît exponentiellement avec  $n$ .

On peut montrer que pour un processus dont les symboles sont tirés uniformément et indépendamment, **en moyenne le codage RLE ne raccourcit pas le message.**

**En français?** Les successions de lettres identiques sont rares, surtout pour  $\geq 3$  lettres identiques.

Pour des **symboles tirés uniformément et indépendamment**, la probabilité d'avoir  $n$  symboles identiques consécutifs décroît exponentiellement avec  $n$ .

On peut montrer que pour un processus dont les symboles sont tirés uniformément et indépendamment, **en moyenne le codage RLE ne raccourcit pas le message.**

**En français?** Les successions de lettres identiques sont rares, surtout pour  $\geq 3$  lettres identiques.

Mais... il existe des motifs qui se répètent.

Pour des **symboles tirés uniformément et indépendamment**, la probabilité d'avoir  $n$  symboles identiques consécutifs décroît exponentiellement avec  $n$ .

On peut montrer que pour un processus dont les symboles sont tirés uniformément et indépendamment, **en moyenne le codage RLE ne raccourcit pas le message.**

**En français?** Les successions de lettres identiques sont rares, surtout pour  $\geq 3$  lettres identiques.

Mais... il existe des motifs qui se répètent.

Il faut donc **transformer le texte** pour obtenir des plages de symboles consécutifs identiques.



Étant donné un texte  $x$ , une façon de créer un autre texte avec des symboles successifs identiques est de **trier** les lettres par ordre alphabétique.

**Problème** : si on ne fait **que trier** les lettres du message, comment **revenir en arrière** ? comment garder l'information de l'ordre initial ?

Étant donné un texte  $x$ , une façon de créer un autre texte avec des symboles successifs identiques est de **trier** les lettres par ordre alphabétique.

**Problème** : si on ne fait **que trier** les lettres du message, comment **revenir en arrière** ? comment garder l'information de l'ordre initial ?

**Exemple.** Prenons le mot : BANANE. On remarque qu'après un A, il y a toujours un N ( $A \rightarrow N$ ).

Étant donné un texte  $x$ , une façon de créer un autre texte avec des symboles successifs identiques est de **trier** les lettres par ordre alphabétique.

**Problème** : si on ne fait **que trier** les lettres du message, comment **revenir en arrière** ? comment garder l'information de l'ordre initial ?

**Exemple.** Prenons le mot : BANANE. On remarque qu'après un A, il y a toujours un N ( $A \rightarrow N$ ).

Cette propriété reste vraie après rotation des lettres :

BANANE, EBANAN, NEBANA, ANEBAN, NANEBA, ANANEB

Étant donné un texte  $x$ , une façon de créer un autre texte avec des symboles successifs identiques est de **trier** les lettres par ordre alphabétique.

**Problème** : si on ne fait **que trier** les lettres du message, comment **revenir en arrière**? comment garder l'information de l'ordre initial?

**Exemple.** Prenons le mot : BANANE. On remarque qu'après un A, il y a toujours un N ( $A \rightarrow N$ ).

Cette propriété reste vraie après rotation des lettres :

BANANE, EBANAN, NEBANA, ANEBAN, NANEBA, ANANEB

Si on trie ces mots par ordre alphabétique :

ANANEB  
ANEBAN  
BANANE  
EBANAN  
NANEBA  
NEBANA

À l'issue du tri, les lettres **N** au début des mots correspondent à des lettres **A** en fin de mot.

De plus, ces lettres sont consécutives.

Si on trie ces mots par ordre alphabétique :

ANANEB  
ANEBAN  
BANANE  
EBANAN  
NANEBA  
NEBAN A

À l'issue du tri, les lettres **N** au début des mots correspondent à des lettres **A** en fin de mot.

De plus, ces lettres sont consécutives.

Si on trie ces mots par ordre alphabétique :

ANANEB  
ANEBAN  
BANANE  
EBANAN  
NANEBA  
NEBANA

À l'issue du tri, les lettres **N** au début des mots correspondent à des lettres **A** en fin de mot.

De plus, ces lettres sont consécutives.

**Idée :** on peut donc aussi utiliser la dernière colonne du tableau comme mot transformé.

Si on trie ces mots par ordre alphabétique :

ANANEB  
ANEBAN  
BANANE  
EBANAN  
NANEBA  
NEBAN A

À l'issue du tri, les lettres **N** au début des mots correspondent à des lettres **A** en fin de mot.

De plus, ces lettres sont consécutives.

**Idée :** on peut donc aussi utiliser la dernière colonne du tableau comme mot transformé.

De plus, il est possible de retrouver **toutes** les colonnes du tableau à partir de la première et de la dernière.

Si on trie ces mots par ordre alphabétique :

ANANEB  
ANEBAN  
BANANE  
EBANAN  
NANEBA  
NEBAN A

À l'issue du tri, les lettres **N** au début des mots correspondent à des lettres **A** en fin de mot.

De plus, ces lettres sont consécutives.

**Idée :** on peut donc aussi utiliser la dernière colonne du tableau comme mot transformé.

De plus, il est possible de retrouver **toutes** les colonnes du tableau à partir de la première et de la dernière.

On peut donc se rappeler de la **position du mot initial** dans la table triée (ici la troisième ligne).



On dispose de

- ▶ une fonction `rotate(mot, i)` qui effectue une rotation d'ordre  $i$  des lettres d'une chaîne de caractères `mot`
- ▶ une fonction `sort(L)` qui trie une liste de chaînes de caractères `L` dans l'ordre alphabétique

## Transformée de Burrows–Wheeler

**Entrée :** une chaîne de caractère  $x = x[0] \dots x[m-1] \in \mathcal{X}^m$ .

**Sortie :** un entier  $n \in \{0, \dots, m-1\}$  et une chaîne de caractères  $y = y[0] \dots y[m-1] \in \mathcal{X}^m$ .

1.  $\text{Tab} \leftarrow [\text{rotate}(x, 0), \dots, \text{rotate}(x, m-1)]$
2.  $\text{SortedTab} \leftarrow \text{sort}(\text{Tab})$
3.  $y \leftarrow [\text{SortedTab}[0][m-1], \dots, \text{SortedTab}[m-1][m-1]]$
4. Trouver  $n \in \{0, \dots, m-1\}$  tel que  $\text{SortedTab}[n] = x$
5. Retourner  $n$  et  $y$

**Complexité :**  $O(m^2)$  opérations pour cette version de l'algorithme, améliorable en  $O(m \log m)$ .

$x = \text{ABRACADABRA.}$

$x = \text{ABRACADABRA.}$

On construit une table  $\text{Tab}$  constituée des rotations cycliques de  $x$ .

$x = \text{ABRACADABRA}$ .

On construit une table  $\text{Tab}$  constituée des rotations cycliques de  $x$ . Puis on trie  $\text{Tab}$  par ordre alphabétique dans  $\text{SortedTab}$ , où

Tab =	ABRACADABRA	et SortedTab =	AABRACADABR
	BRACADABRAA		ABRAABRACAD
	RACADABRAAB		ABRACADABRA
	ACADABRAABR		ACADABRAABR
	CADABRAABRA		ADABRAABRAC
	ADABRAABRAC		BRAABRACADA
	DABRAABRACA		BRACADABRAA
	ABRAABRACAD		CADABRAABRA
	BRAABRACADA		DABRAABRACA
	RAABRACADAB		RAABRACADAB
AABRACADABR	RACADABRAAB		

## Burrows–Wheeler : exemple

$x = \text{ABRACADABRA}$ .

On construit une table `Tab` constituée des rotations cycliques de  $x$ . Puis on trie `Tab` par ordre alphabétique dans `SortedTab`, où

Tab =	ABRACADABRA	et SortedTab =	AABRACADABR
	BRACADABRAA		ABRAABRACAD
	RACADABRAAB		ABRACADABRA
	ACADABRAABR		ACADABRAABR
	CADABRAABRA		ADABRAABRAC
	ADABRAABRAC		BRAABRACADA
	DABRAABRACA		BRACADABRAA
	ABRAABRACAD		CADABRAABRA
	BRAABRACADA		DABRAABRACA
	RAABRACADAB		RAABRACADAB
	AABRACADABR		RACADABRAAB

L'algorithme retourne alors ( $n = 2, y = \text{RDARCAAAAABB}$ ).

## Inversion de la transformée de Burrows–Wheeler

**Entrée :** un entier  $n \in \{0, \dots, m-1\}$  et une chaîne de caractères  $y = y_0 \dots y_{m-1} \in \mathcal{X}^m$ .

**Sortie :** une chaîne de caractère  $x = x_0 \dots x_{m-1} \in \mathcal{X}^m$ .

1.  $\text{Tab} \leftarrow ["" , \dots , "" ]$
2. **Pour tout**  $i \in \{1, \dots, m\}$ 
  - Pour tout**  $j \in \{0, \dots, m-1\}$
  - Ajouter la lettre  $y_j$  au début de la chaîne  $\text{Tab}[j]$
  - $\text{Tab} \leftarrow \text{sort}(\text{Tab})$
3.  $x \leftarrow \text{Tab}[n]$
4. Retourner  $x$

**Complexité :**  $O(m^2 \log m)$  opérations pour cette version de l'algorithme, améliorable en  $O(m \log m)$ .

En sortie d'une transformée de Burrows–Wheeler, on espère observer des motifs de lettres semblables et fréquents.

En sortie d'une transformée de Burrows–Wheeler, on espère observer des motifs de lettres semblables et fréquents.  
On pourrait procéder **directement** à un codage par plage sur la sortie.



En sortie d'une transformée de Burrows–Wheeler, on espère observer des motifs de lettres semblables et fréquents.

On pourrait procéder **directement** à un codage par plage sur la sortie.

Avant, une idée est de **transformer ces motifs en « petits nombres »** grâce à une procédure nommée *move-to-front*.

En sortie d'une transformée de Burrows–Wheeler, on espère observer des motifs de lettres semblables et fréquents.

On pourrait procéder **directement** à un codage par plage sur la sortie.

Avant, une idée est de **transformer ces motifs en « petits nombres »** grâce à une procédure nommée *move-to-front*.

**Avantage** : les petits nombres sont plus courts à stocker (et contiennent plus de 0) que les codes des lettres.

### Algorithme *move-to-front*

**Entrée :** une chaîne de caractère  $x = x_0 \dots x_{m-1} \in \mathcal{X}^m$ .

**Sortie :** une suite d'entiers  $a_0, \dots, a_{m-1} \in \{0, \dots, N-1\}^m$  où  $N = |\mathcal{X}|$

1. **Tab**  $\leftarrow [x_0, \dots, x_{m-1}]$
2. **res**  $\leftarrow []$
3. **Pour tout**  $i \in \{0, \dots, m-1\}$   
    Trouver  $j \in \{0, \dots, m-1\}$  tel que **Tab**[ $j$ ] =  $x_i$   
    **res**  $\leftarrow$  **res** + [ $j$ ]  
    **Tab**  $\leftarrow$  **movefirst**(**Tab**,  $i$ )
4. Retourner **res**

Le logiciel de compression `bzip2`, très courant sous Linux, utilise notamment la séquence d'algorithmes :

`BurrowsWheeler` → `movetofront` → `RLE`

Le logiciel de compression `bzip2`, très courant sous Linux, utilise notamment la séquence d'algorithmes :

`BurrowsWheeler` → `movetofront` → `RLE`

On y ajoute un code de Huffman pour raccourcir le codage des nombres les plus fréquents issus de RLE.