Théorie de l'Information
Cours 9

Julien Lavauzelle

Université Paris 8

Master 1 ACC – Théorie de l'Information

23/11/2022

1. Algorithme de Lempel–Ziv

2. Compression de données

Plan

1. Algorithme de Lempel–Ziv

2. Compression de données

Code universel

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

$$\lim_{n\to\infty} \mathbb{E}_{\boldsymbol{X}} \left[\frac{1}{n} \ell(C_n(X_1, \ldots, X_n)) \right] = \overline{H}(\boldsymbol{X})$$

pour toutes les sources stationnaires $\boldsymbol{X}$.

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.

Présentation

Objectif : construire un code qui soit (presque) universel

Présentation

Objectif : construire un code qui soit (presque) universel

$\Longrightarrow$ algorithmes **LZ77**, **LZ78** de Lempel et Ziv, utilisés comme brique de base dans beaucoup de logiciels de compression actuels.

Présentation

Objectif : construire un code qui soit (presque) universel

$\Longrightarrow$ algorithmes **LZ77**, **LZ78** de Lempel et Ziv, utilisés comme brique de base dans beaucoup de logiciels de compression actuels.

Idée :

1. On parcourt la chaîne de caractères, que l'on découpe en 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).

Présentation

Objectif : construire un code qui soit (presque) universel

$\Longrightarrow$ algorithmes **LZ77**, **LZ78** de Lempel et Ziv, utilisés comme brique de base dans beaucoup de logiciels de compression actuels.

Idée :

1. On parcourt la chaîne de caractères, que l'on découpe en 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 : on considère la séquence sur l'alphabet $\{a, b\}$:

$$aababbabbaabbababbabaabbabaaabbabaaa$$

Présentation

Objectif : construire un code qui soit (presque) universel

$\Longrightarrow$ algorithmes **LZ77**, **LZ78** de Lempel et Ziv, utilisés comme brique de base dans beaucoup de logiciels de compression actuels.

Idée :

1. On parcourt la chaîne de caractères, que l'on découpe en 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 : on considère la séquence sur l'alphabet $\{a, b\}$:

$$aababbabbaabbababbabaabbabaaabbabaaa$$

On peut la découper en :

$$a|\underline{a}b|\underline{ab}b|\underline{abb}a|\underline{abba}b|\underline{abbab}a|\underline{abbaba}a|\underline{abbabaa}a$$

Présentation

Objectif : construire un code qui soit (presque) universel

$\Longrightarrow$ algorithmes **LZ77**, **LZ78** de Lempel et Ziv, utilisés comme brique de base dans beaucoup de logiciels de compression actuels.

Idée :

1. On parcourt la chaîne de caractères, que l'on découpe en 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 : on considère la séquence sur l'alphabet $\{a, b\}$:

$$aababbabbaabbababbabaabbabaaabbabaaa$$

On peut la découper en :

$$a|\underline{a}b|\underline{ab}b|\underline{abb}a|\underline{abba}b|\underline{abbab}a|\underline{abbaba}a|\underline{abbabaa}a$$

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.

Algorithme

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

`res` $= []$
`dict` $= [""]$
`buffer` $= ""$
$i = 0$
Pour j allant de 1 à m :
  Ajouter x_j à la fin de `buffer`
  Si : il existe k tel que `buffer` $=$ `dict`$[k]$
  Alors :
    $i = k$
  Sinon :
    Ajouter `buffer` à `dict`
    Ajouter $(\mathtt{i}, x_j)$ à `res`
    Réinitialiser `buffer` à ""
    Réinitialiser i à 0
Retourner : `res`

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$x = bbaabbabaaaaababbbbaaabbabb$$

`res` $= []$
`dict` $= []$
`buffer` $=$

$$x = |bbaabbabaaaaababbbbaaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaababbbbaaabbabb$$

`res` $= []$
`dict` $= []$
`buffer` $= b$

$$\boldsymbol{x} = |b|baabbabaaaaababbbbaaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaababbbbaaabbabb$$

`res` $= [(0, b)]$
`dict` $= [b]$
`buffer` $= b$

$$\boldsymbol{x} = |b|b|aabbabaaaaababbbbaaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaababbbbaaabbabb$$

`res` $= [(0, b)]$
`dict` $= [b]$
`buffer` $= ba$

$$\boldsymbol{x} = |b|ba|abbabaaaaababbbbaaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaababbbbaaabbabb$$

`res` $= [(0, b), (1, a)]$
`dict` $= [b, ba]$
`buffer` $= a$

$$\boldsymbol{x} = |b|ba|a|bbabaaaaababbbbaaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaababbbbaaabbabb$$

`res` $= [(0, b), (1, a), (0, a)]$
`dict` $= [b, ba, a]$
`buffer` $= b$

$$\boldsymbol{x} = |b|ba|a|b|babaaaaababbbbaaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaababbbbaaabbabb$$

`res` $= [(0,b),(1,a),(0,a)]$
`dict` $= [b, ba, a]$
`buffer` $= bb$

$$\boldsymbol{x} = |b|ba|a|bb|abaaaaababbbbaaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaababbbbaaabbabb$$

`res` $= [(0,b),(1,a),(0,a),(1,b)]$
`dict` $= [b, ba, a, bb]$
`buffer` $= a$

$$\boldsymbol{x} = |b|ba|a|bb|a|baaaaababbbbaaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaababbbbaaabbabb$$

`res` $= [(0,b),(1,a),(0,a),(1,b)]$
`dict` $= [b, ba, a, bb]$
`buffer` $= ab$

$$\boldsymbol{x} = |b|ba|a|bb|ab|aaaaababbbbaaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaababbbbaaabbabb$$

$\texttt{res} = [(0,b),(1,a),(0,a),(1,b),(3,b)]$
$\texttt{dict} = [b, ba, a, bb, ab]$
$\texttt{buffer} = a$

$$\boldsymbol{x} = |b|ba|a|bb|ab|a|aaaababbbbaaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaabababbbbaaabbabb$$

$\texttt{res} = [(0,b),(1,a),(0,a),(1,b),(3,b)]$
$\texttt{dict} = [b, ba, a, bb, ab]$
$\texttt{buffer} = aa$

$$\boldsymbol{x} = |b|ba|a|bb|ab|aa|aaababbbbaaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaabababbbbaaabbabb$$

$\texttt{res} = [(0,b),(1,a),(0,a),(1,b),(3,b),(3,a)]$
$\texttt{dict} = [b, ba, a, bb, ab, aa]$
$\texttt{buffer} = a$

$$\boldsymbol{x} = |b|ba|a|bb|ab|aa|a|aababbbbaaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaababbbbaaabbabb$$

$\texttt{res} = [(0,b),(1,a),(0,a),(1,b),(3,b),(3,a)]$
$\texttt{dict} = [b, ba, a, bb, ab, aa]$
$\texttt{buffer} = aa$

$$\boldsymbol{x} = |b|ba|a|bb|ab|aa|aa|ababbbbaaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaabababbbbaaabbabb$$

$\texttt{res} = [(0,b),(1,a),(0,a),(1,b),(3,b),(3,a)]$
$\texttt{dict} = [b, ba, a, bb, ab, aa]$
$\texttt{buffer} = aaa$

$$\boldsymbol{x} = |b|ba|a|bb|ab|aa|aaa|babbbbaaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaababbbbaaabbabb$$

$\texttt{res} = [(0,b),(1,a),(0,a),(1,b),(3,b),(3,a),(6,a)]$
$\texttt{dict} = [b, ba, a, bb, ab, aa, aaa]$
$\texttt{buffer} = b$

$$\boldsymbol{x} = |b|ba|a|bb|ab|aa|aaa|b|abbbbaaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaabababbbbaaabbabb$$

$\texttt{res} = [(0,b),(1,a),(0,a),(1,b),(3,b),(3,a),(6,a)]$
$\texttt{dict} = [b, ba, a, bb, ab, aa, aaa]$
$\texttt{buffer} = ba$

$$\boldsymbol{x} = |b|ba|a|bb|ab|aa|aaa|ba|bbbbaaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaababbbbaaabbabb$$

$\texttt{res} = [(0,b),(1,a),(0,a),(1,b),(3,b),(3,a),(6,a)]$
$\texttt{dict} = [b, ba, a, bb, ab, aa, aaa]$
$\texttt{buffer} = bab$

$$\boldsymbol{x} = |b|ba|a|bb|ab|aa|aaa|bab|bbbaaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaababbbbaaabbabb$$

$\texttt{res} = [(0,b),(1,a),(0,a),(1,b),(3,b),(3,a),(6,a),(2,b)]$
$\texttt{dict} = [b, ba, a, bb, ab, aa, aaa, bab]$
$\texttt{buffer} = b$

$$\boldsymbol{x} = |b|ba|a|bb|ab|aa|aaa|bab|b|bbaaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaababbbbaaabbabb$$

$\texttt{res} = [(0,b),(1,a),(0,a),(1,b),(3,b),(3,a),(6,a),(2,b)]$
$\texttt{dict} = [b, ba, a, bb, ab, aa, aaa, bab]$
$\texttt{buffer} = bb$

$$\boldsymbol{x} = |b|ba|a|bb|ab|aa|aaa|bab|bb|baaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaabababbbbaaabbabb$$

$\texttt{res} = [(0,b),(1,a),(0,a),(1,b),(3,b),(3,a),(6,a),(2,b)]$
$\texttt{dict} = [b, ba, a, bb, ab, aa, aaa, bab]$
$\texttt{buffer} = bbb$

$$\boldsymbol{x} = |b|ba|a|bb|ab|aa|aaa|bab|bbb|aaabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaabababbbbaaabbabb$$

$\texttt{res} = [(0,b),(1,a),(0,a),(1,b),(3,b),(3,a),(6,a),(2,b),(4,b)]$
$\texttt{dict} = [b, ba, a, bb, ab, aa, aaa, bab, bbb]$
$\texttt{buffer} = a$

$$\boldsymbol{x} = |b|ba|a|bb|ab|aa|aaa|bab|bbb|a|aabbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaabababbbbaaabbabb$$

$\texttt{res} = [(0,b),(1,a),(0,a),(1,b),(3,b),(3,a),(6,a),(2,b),(4,b)]$
$\texttt{dict} = [b, ba, a, bb, ab, aa, aaa, bab, bbb]$
$\texttt{buffer} = aa$

$$\boldsymbol{x} = |b|ba|a|bb|ab|aa|aaa|bab|bbb|aa|abbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaabababbbbaaabbabb$$

$\texttt{res} = [(0,b),(1,a),(0,a),(1,b),(3,b),(3,a),(6,a),(2,b),(4,b)]$
$\texttt{dict} = [b, ba, a, bb, ab, aa, aaa, bab, bbb]$
$\texttt{buffer} = aaa$

$$\boldsymbol{x} = |b|ba|a|bb|ab|aa|aaa|bab|bbb|aaa|bbabb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaabababbbbaaabbabb$$

`res` $= [(0,b),(1,a),(0,a),(1,b),(3,b),(3,a),(6,a),(2,b),(4,b)]$
`dict` $= [b, ba, a, bb, ab, aa, aaa, bab, bbb]$
`buffer` $= aaab$

$$\boldsymbol{x} = |b|ba|a|bb|ab|aa|aaa|bab|bbb|aaab|babb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaababbbbaaabbabb$$

$\texttt{res} = [(0,b),(1,a),(0,a),(1,b),(3,b),(3,a),(6,a),(2,b),(4,b),(7,a)]$
$\texttt{dict} = [b, ba, a, bb, ab, aa, aaa, bab, bbb, aaab]$
$\texttt{buffer} = b$

$$\boldsymbol{x} = |b|ba|a|bb|ab|aa|aaa|bab|bbb|aaab|b|abb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaabababbbbaaabbabb$$

$\texttt{res} = [(0,b),(1,a),(0,a),(1,b),(3,b),(3,a),(6,a),(2,b),(4,b),(7,a)]$
$\texttt{dict} = [b, ba, a, bb, ab, aa, aaa, bab, bbb, aaab]$
$\texttt{buffer} = ba$

$$\boldsymbol{x} = |b|ba|a|bb|ab|aa|aaa|bab|bbb|aaab|ba|bb$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaabababbbbaaabbabb$$

$\texttt{res} = [(0,b),(1,a),(0,a),(1,b),(3,b),(3,a),(6,a),(2,b),(4,b),(7,a)]$
$\texttt{dict} = [b, ba, a, bb, ab, aa, aaa, bab, bbb, aaab]$
$\texttt{buffer} = bab$

$$\boldsymbol{x} = |b|ba|a|bb|ab|aa|aaa|bab|bbb|aaab|bab|b$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaabababbbbaaabbabb$$

$\texttt{res} = [(0,b),(1,a),(0,a),(1,b),(3,b),(3,a),(6,a),(2,b),(4,b),(7,a)]$
$\texttt{dict} = [b, ba, a, bb, ab, aa, aaa, bab, bbb, aaab]$
$\texttt{buffer} = babb$

$$\boldsymbol{x} = |b|ba|a|bb|ab|aa|aaa|bab|bbb|aaab|babb|$$

Mise en application

Algorithme de Lempel-Ziv-Welch (LZW)

La chaîne en entrée est :

$$\boldsymbol{x} = bbaabbabaaaaabababbbbaaabbabb$$

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

$$\boldsymbol{x} = |b|ba|a|bb|ab|aa|aaa|bab|bbb|aaab|babb|$$

Analyse de LZW

Question :

Ce codage est-il « bon » ?

Analyse de LZW

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 stochasique donné, quelle est la longueur moyenne du codage LZW ?

Analyse : pire cas

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

Analyse : pire cas

Objectif : trouver une chaîne de caractères $\boldsymbol{x}$ pour laquelle le codage LZW est asymptotiquement le plus long.

$\Longrightarrow$ Faut-il que $\boldsymbol{x}$ soit divisée en mots petits ? en mots ressemblants et longs ?

Analyse : pire cas

Objectif : trouver une chaîne de caractères $\boldsymbol{x}$ pour laquelle le codage LZW est asymptotiquement le plus long.

$\Longrightarrow$ Faut-il que $\boldsymbol{x}$ soit divisée en mots petits ? en mots ressemblants et longs ?

Rappel : l'exemple typique donne un codage assez court

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

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

Analyse : pire cas

Objectif : trouver une chaîne de caractères $\boldsymbol{x}$ pour laquelle le codage LZW est asymptotiquement le plus long.

$\Longrightarrow$ Faut-il que $\boldsymbol{x}$ soit divisée en mots petits ? en mots ressemblants et longs ?

Rappel : l'exemple typique donne un codage assez court

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

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

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

donne le mot

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

Analyse : pire cas

Objectif : trouver une chaîne de caractères $\boldsymbol{x}$ pour laquelle le codage LZW est asymptotiquement le plus long.

$\Longrightarrow$ Faut-il que $\boldsymbol{x}$ soit divisée en mots petits ? en mots ressemblants et longs ?

Rappel : l'exemple typique donne un codage assez court

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

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

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

donne le mot

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

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

Analyse : pire cas

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

Analyse : pire cas

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

On montre que $\ell(\boldsymbol{x}^{(k)}) = (k-1)2^{k+1} + 2$ — **preuve :** exercice.

Analyse : pire cas

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

On montre que $\ell(\boldsymbol{x}^{(k)}) = (k-1)2^{k+1} + 2$ — **preuve :** exercice.

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

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

Analyse : pire cas

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

On montre que $\ell(\boldsymbol{x}^{(k)}) = (k-1)2^{k+1} + 2$ — **preuve :** exercice.

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

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

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

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

Analyse : pire cas

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

On montre que $\ell(\boldsymbol{x}^{(k)}) = (k-1)2^{k+1} + 2$ — **preuve :** exercice.

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

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

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

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

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

$$\frac{\ell(\boldsymbol{c}^{(k)})}{\ell(\boldsymbol{x}^{(k)})} = \left(1 + \frac{2}{k-1}\right)\left(1 + \frac{2}{\ell(\boldsymbol{x}^{(k)})}\right) - \frac{2}{\ell(\boldsymbol{x}^{(k)})} \underset{n\to\infty}{=} 1 + o\left(\frac{1}{\log_2 n}\right)\,.$$

Analyse : pire cas

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

On montre que $\ell(\boldsymbol{x}^{(k)}) = (k-1)2^{k+1} + 2$ — **preuve :** exercice.

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

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

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

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

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

$$\frac{\ell(\boldsymbol{c}^{(k)})}{\ell(\boldsymbol{x}^{(k)})} = \left(1 + \frac{2}{k-1}\right)\left(1 + \frac{2}{\ell(\boldsymbol{x}^{(k)})}\right) - \frac{2}{\ell(\boldsymbol{x}^{(k)})} \underset{n\to\infty}{=} 1 + o\left(\frac{1}{\log_2 n}\right)\,.$$

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

Analyse : cas moyen

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

Processus **ergodique** : les statistiques de la source (par exemple, son entropie) peuvent être approchées à partir d'une réalisation suffisamment longue.

Analyse : cas moyen

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

Processus **ergodique** : les statistiques de la source (par exemple, son entropie) peuvent être approchées à partir d'une réalisation suffisamment longue.

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

$$\lim_{n \to +\infty} \frac{1}{n} \ell(C(\boldsymbol{x}^n)) = \overline{H}(\boldsymbol{X}) \,.$$

Analyse : cas moyen

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

Processus **ergodique** : les statistiques de la source (par exemple, son entropie) peuvent être approchées à partir d'une réalisation suffisamment longue.

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

$$\lim_{n \to +\infty} \frac{1}{n} \ell(C(\boldsymbol{x}^n)) = \overline{H}(\boldsymbol{X}) \,.$$

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**.

Plan

1. Algorithme de Lempel–Ziv

2. Compression de données

Motivation

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

Motivation

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

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

Motivation

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%

Motivation

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 ?

Exemples

$$\boldsymbol{x} = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbb$$

Exemples

$$\boldsymbol{x} = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbb$$

$$C(\boldsymbol{x}) = 29a7b$$

Exemples

$$\boldsymbol{x} = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbb$$

$$C(\boldsymbol{x}) = 29a7b$$

$$\boldsymbol{y} = abababababababababababababab$$

Exemples

$$\boldsymbol{x} = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbb$$

$$C(\boldsymbol{x}) = 29a7b$$

$$\boldsymbol{y} = abababababababababababababab$$

$$C(\boldsymbol{y}) = 14(ab)$$

Exemples

$$\boldsymbol{x} = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbb$$

$$C(\boldsymbol{x}) = 29a7b$$

$$\boldsymbol{y} = abababababababababababababab$$

$$C(\boldsymbol{y}) = 14(ab)$$

$$\boldsymbol{z} = abcdefghijklmnopabcdefghijklmnopabcdefghijklmnop$$

Exemples

$$\boldsymbol{x} = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbb$$

$$C(\boldsymbol{x}) = 29a7b$$

$$\boldsymbol{y} = abababababababababababababab$$

$$C(\boldsymbol{y}) = 14(ab)$$

$$\boldsymbol{z} = abcdefghijklmnopabcdefghijklmnopabcdefghijklmnop$$

$$C(\boldsymbol{z}) = 3(abcdefghijklmnop)$$

Exemples

$$\boldsymbol{x} = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbb$$

$$C(\boldsymbol{x}) = 29a7b$$

$$\boldsymbol{y} = abababababababababababababab$$

$$C(\boldsymbol{y}) = 14(ab)$$

$$\boldsymbol{z} = abcdefghijklmnopabcdefghijklmnopabcdefghijklmnop$$

$$C(\boldsymbol{z}) = 3(abcdefghijklmnop)$$

Question : Comment automatiser ces procédures ?

RLE

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

RLE

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.

RLE

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 :

$$\boldsymbol{x} = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbb \quad \mapsto \quad C(\boldsymbol{x}) = 29a7b\,.$$

RLE

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 :

$$\boldsymbol{x} = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbb \quad \mapsto \quad C(\boldsymbol{x}) = 29a7b\,.$$

Sur $\boldsymbol{y} = abababababababababababababab$, on obtient

$$C(\boldsymbol{y}) = 1a1b1a1b1a1b1a1b1a1b1a1b1a1b1a1b1a1b1a1b1a1b1a1b1a1b1a1b1a1b\,.$$

RLE : cas limites

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

RLE : cas limites

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

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

RLE : cas limites

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 :

RLE : cas limites

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 : $n \times s$

RLE : cas limites

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 :

RLE : cas limites

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 : $\lceil \log_2 n \rceil + s$

RLE : cas limites

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.

RLE : cas limites

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 :

RLE : cas limites

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 : $n \times s$

RLE : cas limites

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 :

RLE : cas limites

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 : $n(1 + s)$

RLE : cas moyen

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

RLE : cas moyen

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**.

RLE : cas moyen

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 ≥ 3 lettres identiques.

RLE : cas moyen

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 ≥ 3 lettres identiques.

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

RLE : cas moyen

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 ≥ 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.

Burrows-Wheeler : idée

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 : comment revenir en arrière ? comment garder l'information de l'ordre initial ?

Burrows-Wheeler : idée

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 : comment revenir en arrière ? comment garder l'information de l'ordre initial ?

Prenons le mot : BANANE.

On remarque qu'après un A, il y a toujours un N (A $\rightarrow$ N).

Burrows-Wheeler : idée

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 : comment revenir en arrière ? comment garder l'information de l'ordre initial ?

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

Burrows-Wheeler : idée

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 : comment revenir en arrière ? comment garder l'information de l'ordre initial ?

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

Les lettres N consécutives issues du tri correspondent à des lettres A consécutives en fin de mot (car A $\rightarrow$ N).

Burrows-Wheeler : idée

Si on trie ces mots par ordre alphabétique :

ANANEB
ANEBAN
BANANE
EBANAN
NANEBA
NEBANA

Les lettres N consécutives issues du tri correspondent à des lettres A consécutives en fin de mot (car A $\rightarrow$ N).

Burrows-Wheeler : idée

Si on trie ces mots par ordre alphabétique :

ANANEB
ANEBAN
BANANE
EBANAN
NANEBA
NEBANA

Les lettres N consécutives issues du tri correspondent à des lettres A consécutives en fin de mot (car A $\rightarrow$ N).

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

Burrows-Wheeler : idée

Si on trie ces mots par ordre alphabétique :

ANANEB
ANEBAN
BANANE
EBANAN
NANEBA
NEBANA

Les lettres N consécutives issues du tri correspondent à des lettres A consécutives en fin de mot (car A $\rightarrow$ N).

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.

Burrows-Wheeler : idée

Si on trie ces mots par ordre alphabétique :

ANANEB
ANEBAN
BANANE
EBANAN
NANEBA
NEBANA

Les lettres N consécutives issues du tri correspondent à des lettres A consécutives en fin de mot (car A $\rightarrow$ N).

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).

Burrows-Wheeler : algorithme

On dispose de

- une fonction de rotation des lettres `rotate`
- une fonction de tri `sort`

Transformée de Burrows-Wheeler

Entrée : une chaîne de caractère $\boldsymbol{x} = x_0 \dots x_{m-1} \in \mathcal{X}^m$.
Sortie : un entier $n \in \{0, \dots, m-1\}$ et une chaîne de caractères $\boldsymbol{y} = y_0 \dots y_{m-1} \in \mathcal{X}^m$.

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

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

Burrows-Wheeler : exemple

$$x = \texttt{ABRACADABRA}.$$

Burrows-Wheeler : exemple

$$x = \texttt{ABRACADABRA}.$$

On construit une table `Tab` constituée des rotations cycliques de x.

Burrows-Wheeler : exemple

$$x = \texttt{ABRACADABRA}.$$

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

$$\texttt{Tab} = \begin{bmatrix} \texttt{ABRACADABRA} \\ \texttt{BRACADABRAA} \\ \texttt{RACADABRAAB} \\ \texttt{ACADABRAABR} \\ \texttt{CADABRAABRA} \\ \texttt{ADABRAABRAC} \\ \texttt{DABRAABRACA} \\ \texttt{ABRAABRACAD} \\ \texttt{BRAABRACADA} \\ \texttt{RAABRACADAB} \\ \texttt{AABRACADABR} \end{bmatrix} \quad \text{et} \quad \texttt{SortedTab} = \begin{bmatrix} \texttt{AABRACADABR} \\ \texttt{ABRAABRACAD} \\ \texttt{ABRACADABRA} \\ \texttt{ACADABRAABR} \\ \texttt{ADABRAABRAC} \\ \texttt{BRAABRACADA} \\ \texttt{BRACADABRAA} \\ \texttt{CADABRAABRA} \\ \texttt{DABRAABRACA} \\ \texttt{RAABRACADAB} \\ \texttt{RACADABRAAB} \end{bmatrix}.$$

Burrows-Wheeler : exemple

$$x = \texttt{ABRACADABRA}.$$

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

$$\texttt{Tab} = \begin{bmatrix} \texttt{ABRACADABRA} \\ \texttt{BRACADABRAA} \\ \texttt{RACADABRAAB} \\ \texttt{ACADABRAABR} \\ \texttt{CADABRAABRA} \\ \texttt{ADABRAABRAC} \\ \texttt{DABRAABRACA} \\ \texttt{ABRAABRACAD} \\ \texttt{BRAABRACADA} \\ \texttt{RAABRACADAB} \\ \texttt{AABRACADABR} \end{bmatrix} \quad \text{et} \quad \texttt{SortedTab} = \begin{bmatrix} \texttt{AABRACADABR} \\ \texttt{ABRAABRACAD} \\ \texttt{ABRACADABRA} \\ \texttt{ACADABRAABR} \\ \texttt{ADABRAABRAC} \\ \texttt{BRAABRACADA} \\ \texttt{BRACADABRAA} \\ \texttt{CADABRAABRA} \\ \texttt{DABRAABRACA} \\ \texttt{RAABRACADAB} \\ \texttt{RACADABRAAB} \end{bmatrix}.$$

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

Inversion de la transformée de Burrows-Wheeler

Entrée :un entier $n \in \{0, \ldots, m-1\}$ et une chaîne de caractères $\boldsymbol{y} = y_0 \ldots y_{m-1} \in \mathcal{X}^m$.
Sortie : une chaîne de caractère $\boldsymbol{x} = x_0 \ldots x_{m-1} \in \mathcal{X}^m$.

1. `Tab` $\leftarrow [y_0, y_1, \ldots, y_{m-1}]$
2. `Tab` $\leftarrow$ `sort`(`Tab`)
3. **Pour tout** $i \in \{1, \ldots, m-1\}$
   Pour tout $j \in \{0, \ldots, m-1\}$
   Ajouter y_j au début de la chaîne `Tab[j]`
   `Tab` $\leftarrow$ `sort`(`Tab`)
4. $\boldsymbol{x} \leftarrow$ `Tab`[`n`]
5. Retourner $\boldsymbol{x}$

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

move-to-front : motivation

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

move-to-front : motivation

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.

move-to-front : motivation

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*.

move-to-front : motivation

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 $\boldsymbol{x} = x_0 \dots x_{m-1} \in \mathcal{X}^m$.
Sortie : une suite d'entiers $a_0, \dots, a_{m-1} \in \{0, \dots, N-1\}^m$ où $N = |\mathcal{X}|$

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

Conclusion

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

$$\texttt{BurrowsWheeler} \quad \rightarrow \quad \texttt{movetofront} \quad \rightarrow \quad \texttt{RLE}$$

Conclusion

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

$$\texttt{BurrowsWheeler} \quad \rightarrow \quad \texttt{movetofront} \quad \rightarrow \quad \texttt{RLE}$$

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