Algorithme de Lempel-Ziv-Welch

Lycée Thiers

Algorithme de Lempel-Ziv-Welch

Lycée Thiers

Crédits

- Un cours de Marc de Falco.
- Wikipedia
- Informatique -Cours et exercices corrigés- (MP2I-MPI) (ellipse)

Présentation

- L'algorithme de *Lempel-Ziv-Welch* (LZW)est un algorithme de compression de données sans perte.

Présentation

- L'algorithme de *Lempel-Ziv-Welch* (LZW)est un algorithme de compression de données sans perte.
- Ses inventeurs sont Abraham Lempel, Jakob Ziv qui l'ont proposé en 1977 et Terry Welch qui l'a finalisé en 1984.

Présentation

- L'algorithme de *Lempel-Ziv-Welch* (LZW)est un algorithme de compression de données sans perte.
- Ses inventeurs sont Abraham Lempel, Jakob Ziv qui l'ont proposé en 1977 et Terry Welch qui l'a finalisé en 1984.
- LZW a été utilisé dans des modems aujourd'hui obsolètes mais on le trouve encore dans la compression des images « GIFF » ou « TIFF » et les fichiers audio « MOD ». Il est à la base de la compression « ZIP ».

Présentation

- L'algorithme de *Lempel-Ziv-Welch* (LZW)est un algorithme de compression de données sans perte.
- Ses inventeurs sont Abraham Lempel, Jakob Ziv qui l'ont proposé en 1977 et Terry Welch qui l'a finalisé en 1984.
- LZW a été utilisé dans des modems aujourd'hui obsolètes mais on le trouve encore dans la compression des images « GIFF » ou « TIFF » et les fichiers audio « MOD ». Il est à la base de la compression « ZIP ».
- Facile à coder (c'est son principal avantage) il n'est souvent pas optimal car il n'effectue qu'une analyse sommaire des données à compresser.

Principe

- Rechercher dans le texte à compresser des répétitions de sous-chaînes identiques et leur donner une forme compacte dans le texte compressé.

Principe

- Rechercher dans le texte à compresser des répétitions de sous-chaînes identiques et leur donner une forme compacte dans le texte compressé.
- L'algorithme LZW procède en une seule passe, en maintenant, au fur et à mesure de la compression, l'ensemble des *facteurs* qu'il a déjà rencontrés.

Principe

- Rechercher dans le texte à compresser des répétitions de sous-chaînes identiques et leur donner une forme compacte dans le texte compressé.
- L'algorithme LZW procède en une seule passe, en maintenant, au fur et à mesure de la compression, l'ensemble des *facteurs* qu'il a déjà rencontrés.
- Cette caractéristique est adaptée à la compression d'un texte qu'on découvre à la volée comme lorsque le texte est transmis via un canal de communication.

Préfixe, suffixe

Définition

- Le mot x est appelé un *préfixe* du mot m si il existe un mot y tel que $m = x \cdot y$.

Exemple

Préfixe, suffixe

Définition

- Le mot x est appelé un *préfixe* du mot m si il existe un mot y tel que $m = x \cdot y$.
- Le mot x est appelé un *suffixe* du mot m si il existe un mot y tel que $m = y \cdot x$.

Exemple

Préfixe, suffixe

Définition

- Le mot x est appelé un *préfixe* du mot m si il existe un mot y tel que $m = x \cdot y$.
- Le mot x est appelé un *suffixe* du mot m si il existe un mot y tel que $m = y \cdot x$.

Exemple

- ε, **langage** et **lang** sont des préfixes de **langage**,

Préfixe, suffixe

Définition

- Le mot x est appelé un *préfixe* du mot m si il existe un mot y tel que $m = x \cdot y$.
- Le mot x est appelé un *suffixe* du mot m si il existe un mot y tel que $m = y \cdot x$.

Exemple

- ε, **langage** et **lang** sont des préfixes de **langage**,
- ε, **langage** et **gage** sont des suffixes de **langage**,

Préfixe, suffixe

Définition

- Le mot x est appelé un *préfixe* du mot m si il existe un mot y tel que $m = x \cdot y$.
- Le mot x est appelé un *suffixe* du mot m si il existe un mot y tel que $m = y \cdot x$.

Exemple

- ε, **langage** et **lang** sont des préfixes de **langage**,
- ε, **langage** et **gage** sont des suffixes de **langage**,
- Si $xu = m$ et $xv = m$ alors, par régularité, $u = v$.

Facteurs

Définition

On dit qu'un mot x est *facteur* d'un mot m s'il existe u, v, deux mots tels que $m = uxv$.
Le mot $x = x_1 \dots x_n$ où les x_i sont des caractères est un *sous-mot* de m s'il existe $n-1$ mots $u_1, \dots, u_{n-1}$ tels que $x_1 u_1 x_2 u_2 \dots x_{n-1} u_{n-1} x_n$ est un facteur de m.

Exemple

Le mot **sol** est facteur de **insolent**. **ilet** est un sous-mot de **insolent**.

Pour plus d'informations sur la théorie des mots, voir par exemple ce cours.

Table des correspondances facteurs/encodage

- L'algorithme de compression construit une table de traduction des facteurs du texte en parcourant le texte à compresser.

Table des correspondances facteurs/encodage

- L'algorithme de compression construit une table de traduction des facteurs du texte en parcourant le texte à compresser.
- Cette table relie des codes de taille (le plus souvent) fixée (généralement à 12 bits) aux chaînes de caractères. Certaines implémentations avec taille d'encodage variable existent aussi.

Table des correspondances facteurs/encodage

- L'algorithme de compression construit une table de traduction des facteurs du texte en parcourant le texte à compresser.
- Cette table relie des codes de taille (le plus souvent) fixée (généralement à 12 bits) aux chaînes de caractères. Certaines implémentations avec taille d'encodage variable existent aussi.
- La table est initialisée avec tous les caractères (256 entrées dans le cas de caractères codés sur 8 bits). C'est une injection qui associe une valeur numérique à tout caractère de l'alphabet.

Table des correspondances facteurs/encodage

- L'algorithme de compression construit une table de traduction des facteurs du texte en parcourant le texte à compresser.
- Cette table relie des codes de taille (le plus souvent) fixée (généralement à 12 bits) aux chaînes de caractères. Certaines implémentations avec taille d'encodage variable existent aussi.
- La table est initialisée avec tous les caractères (256 entrées dans le cas de caractères codés sur 8 bits). C'est une injection qui associe une valeur numérique à tout caractère de l'alphabet.
- Il est malin d'utiliser un dictionnaire (facteur, encodage). Les seules clés du dictionnaires qui ne sont pas des facteurs du texte sont les caractères de l'alphabet non utilisés par le texte.

Table des correspondances facteurs/encodage

- L'algorithme de compression construit une table de traduction des facteurs du texte en parcourant le texte à compresser.
- Cette table relie des codes de taille (le plus souvent) fixée (généralement à 12 bits) aux chaînes de caractères. Certaines implémentations avec taille d'encodage variable existent aussi.
- La table est initialisée avec tous les caractères (256 entrées dans le cas de caractères codés sur 8 bits). C'est une injection qui associe une valeur numérique à tout caractère de l'alphabet.
- Il est malin d'utiliser un dictionnaire (facteur, encodage). Les seules clés du dictionnaires qui ne sont pas des facteurs du texte sont les caractères de l'alphabet non utilisés par le texte.
- L'algorithme LZW exploite et modifie à la volée le dictionnaire des facteurs. Il renvoie une liste de clés de ce dictionnaire (c'est à dire une liste d'entiers), chacune codant un facteur du texte.

Algorithme

Listing 1 – Algorithme LZW

```
1  /*L'alphabet Σ est supposé connu.*/
2  fonction lzw_compress(t : texte):
3      initialiser d avec Σ; /* dictionnaire (facteur,code) */
4      w ← ε; /*le facteur courant*/
5      t' ← liste vide; /*accumulateur de codes*/
6      n ← |Σ|; /*nombre de facteurs déjà compressés*/
7      i ← 0 /*position dans t*/
8      tant_que i < |t| faire :
9          c ← t[i]; i++;/*déplacer le curseur de lecture*/
10         p ← w + c;/*ajouter une lettre à w*/
11         si p est une clé de d:
12             w ← p;
13         sinon:
14             d[p] ← n;/*ajouter l'association (p,n)*/
15             n++;/*incrémenter le nb de codes enregistrés*/
16             /*rqe : n = |d| : nb de clés dans le dico*/
17             append t' d[w]; /*ajouter un nouveau code*/
18             w ← c;
19     append t' d[w];
20     renvoyer inverse de t';
```

Invariant

Avec les conventions du code ci-dessus :
Si on considère (artificiellement) que ε est une clé du dictionnaire (par exemple encodée par -1) alors « *w est une clé du dictionnaire* » est un invariant de boucle.

Exemple (Wikipedia)

On veut compresser "**TOBEORNOTTOBEORTOBEORNOT**".

- Initialisation de d : (**A** :65) ... (**T** :84), (**O** :79), (**B** :66), (**E** :69), (**R** :82), (**N** :78) ... (**Z** :90) ... (\255,255) et $t' \leftarrow \varepsilon$ (texte compressé)

Exemple (Wikipedia)

On veut compresser "**TOBEORNOTTOBEORTOBEORNOT**".

- Initialisation de d : (**A** :65) ... (**T** :84), (**O** :79), (**B** :66), (**E** :69), (**R** :82), (**N** :78) ... (**Z** :90) ... (\255,255) et $t' \leftarrow \varepsilon$ (texte compressé)
- Position 0 : **T** est une clé mais pas **TO**. $d[\mathbf{TO}] \leftarrow 255 + 1 = 256$, $t' \leftarrow 84$

Exemple (Wikipedia)

On veut compresser "**TOBEORNOTTOBEORTOBEORNOT**".

- Initialisation de d : (**A** :65) ... (**T** :84), (**O** :79), (**B** :66), (**E** :69), (**R** :82), (**N** :78) ... (**Z** :90) ... (\255,255) et $t' \leftarrow \varepsilon$ (texte compressé)
- Position 0 : **T** est une clé mais pas **TO**. $d[\mathbf{TO}] \leftarrow 255 + 1 = 256$, $t' \leftarrow 84$
- Position 1 : **O** est une clé mais pas **OB**. $d[\mathbf{OB}] \leftarrow 257$, $t' \leftarrow 84, 79$

Exemple (Wikipedia)

On veut compresser "**TOBEORNOTTOBEORTOBEORNOT**".

- Initialisation de d : (**A** :65) ... (**T** :84), (**O** :79), (**B** :66), (**E** :69), (**R** :82), (**N** :78) ... (**Z** :90) ... (\255,255) et $t' \leftarrow \varepsilon$ (texte compressé)
- Position 0 : **T** est une clé mais pas **TO**. $d[\mathbf{TO}] \leftarrow 255 + 1 = 256$, $t' \leftarrow 84$
- Position 1 : **O** est une clé mais pas **OB**. $d[\mathbf{OB}] \leftarrow 257$, $t' \leftarrow 84, 79$
- Position 2 : **B** est une clé mais pas **BE**. $d[\mathbf{BE}] \leftarrow 258$, $t' \leftarrow 84, 79, 66$

Exemple (Wikipedia)

On veut compresser "**TOBEORNOTTOBEORTOBEORNOT**".

- Initialisation de d : (**A** :65) ... (**T** :84), (**O** :79), (**B** :66), (**E** :69), (**R** :82), (**N** :78) ... (**Z** :90) ... (\255,255) et $t' \leftarrow \varepsilon$ (texte compressé)
- Position 0 : **T** est une clé mais pas **TO**. $d[\mathbf{TO}] \leftarrow 255 + 1 = 256$, $t' \leftarrow 84$
- Position 1 : **O** est une clé mais pas **OB**. $d[\mathbf{OB}] \leftarrow 257$, $t' \leftarrow 84, 79$
- Position 2 : **B** est une clé mais pas **BE**. $d[\mathbf{BE}] \leftarrow 258$, $t' \leftarrow 84, 79, 66$
- Position 3 : **E** est une clé mais pas **EO**. $d[\mathbf{EO}] \leftarrow 259$, $t' \leftarrow 84, 79, 66, 69$

Exemple (Wikipedia)

On veut compresser "**TOBEORNOTTOBEORTOBEORNOT**".

- Initialisation de d : (**A** :65) ... (**T** :84), (**O** :79), (**B** :66), (**E** :69), (**R** :82), (**N** :78) ... (**Z** :90) ... (\255,255) et $t' \leftarrow \varepsilon$ (texte compressé)
- Position 0 : **T** est une clé mais pas **TO**. $d[\mathbf{TO}] \leftarrow 255 + 1 = 256$, $t' \leftarrow 84$
- Position 1 : **O** est une clé mais pas **OB**. $d[\mathbf{OB}] \leftarrow 257$, $t' \leftarrow 84, 79$
- Position 2 : **B** est une clé mais pas **BE**. $d[\mathbf{BE}] \leftarrow 258$, $t' \leftarrow 84, 79, 66$
- Position 3 : **E** est une clé mais pas **EO**. $d[\mathbf{EO}] \leftarrow 259$, $t' \leftarrow 84, 79, 66, 69$
- Position 4 : **O** est une clé mais pas **OR**. $d[\mathbf{OR}] \leftarrow 260$, $t' \leftarrow 84, 79, 66, 69, 79$

Exemple (Wikipedia)

On veut compresser "**TOBEORNOTTOBEORTOBEORNOT**".

- Initialisation de d : (**A** :65) ... (**T** :84), (**O** :79), (**B** :66), (**E** :69), (**R** :82), (**N** :78) ... (**Z** :90) ... (\255,255) et $t' \leftarrow \varepsilon$ (texte compressé)
- Position 0 : **T** est une clé mais pas **TO**. $d[\mathbf{TO}] \leftarrow 255 + 1 = 256$, $t' \leftarrow 84$
- Position 1 : **O** est une clé mais pas **OB**. $d[\mathbf{OB}] \leftarrow 257$, $t' \leftarrow 84, 79$
- Position 2 : **B** est une clé mais pas **BE**. $d[\mathbf{BE}] \leftarrow 258$, $t' \leftarrow 84, 79, 66$
- Position 3 : **E** est une clé mais pas **EO**. $d[\mathbf{EO}] \leftarrow 259$, $t' \leftarrow 84, 79, 66, 69$
- Position 4 : **O** est une clé mais pas **OR**. $d[\mathbf{OR}] \leftarrow 260$, $t' \leftarrow 84, 79, 66, 69, 79$
- Position 5 : **R** est une clé mais pas **RN**. $d[\mathbf{RN}] \leftarrow 261$, Puis **N** est une clé mais pas **NO**. $t' \leftarrow 84, 79, 66, 69, 79, 82, 78$

Exemple (Wikipedia)

On veut compresser "**TOBEORNOTTOBEORTOBEORNOT**".

- Position 7 : **O** est une clé mais pas **OT**. $d[\mathbf{OT}] \leftarrow 263$, $t' \leftarrow 84, 79, 66, 69, 79, 82, 78, 79$. **T** est une clé mais pas **TT**. $d[\mathbf{TT}] \leftarrow 264$, $t' \leftarrow 84, 79, 66, 69, 79, 82, 78, 79, 84$

Exemple (Wikipedia)

On veut compresser "**TOBEORNOTTOBEORTOBEORNOT**".

- Position 7 : **O** est une clé mais pas **OT**. $d[\mathbf{OT}] \leftarrow 263$, $t' \leftarrow 84, 79, 66, 69, 79, 82, 78, 79$. **T** est une clé mais pas **TT**. $d[\mathbf{TT}] \leftarrow 264$, $t' \leftarrow 84, 79, 66, 69, 79, 82, 78, 79, 84$
- Position 9 : **T**, **TO** sont des clés mais pas **TOB**. $d[\mathbf{TOB}] \leftarrow 265$, $t' \leftarrow 84, 79, 66, 69, 79, 82, 78, 79, 84, 256$

Exemple (Wikipedia)

On veut compresser "**TOBEORNOTTOBEORTOBEORNOT**".

- Position 7 : **O** est une clé mais pas **OT**. $d[\mathbf{OT}] \leftarrow 263$, $t' \leftarrow 84, 79, 66, 69, 79, 82, 78, 79$. **T** est une clé mais pas **TT**. $d[\mathbf{TT}] \leftarrow 264$, $t' \leftarrow 84, 79, 66, 69, 79, 82, 78, 79, 84$
- Position 9 : **T**, **TO** sont des clés mais pas **TOB**. $d[\mathbf{TOB}] \leftarrow 265$, $t' \leftarrow 84, 79, 66, 69, 79, 82, 78, 79, 84, 256$
- Position 11 : **B**, **BE** sont des clés mais pas **BEO**. $d[\mathbf{BEO}] \leftarrow 266$, $t' \leftarrow 84, 79, 66, 69, 79, 82, 78, 79, 84, 256, 258$ etc.

Exemple (Wikipedia)

On veut compresser "**TOBEORNOTTOBEORTOBEORNOT**".

- Position 7 : **O** est une clé mais pas **OT**. $d[\mathbf{OT}] \leftarrow 263$, $t' \leftarrow 84, 79, 66, 69, 79, 82, 78, 79$. **T** est une clé mais pas **TT**. $d[\mathbf{TT}] \leftarrow 264$, $t' \leftarrow 84, 79, 66, 69, 79, 82, 78, 79, 84$
- Position 9 : **T**, **TO** sont des clés mais pas **TOB**. $d[\mathbf{TOB}] \leftarrow 265$, $t' \leftarrow 84, 79, 66, 69, 79, 82, 78, 79, 84, 256$
- Position 11 : **B**, **BE** sont des clés mais pas **BEO**. $d[\mathbf{BEO}] \leftarrow 266$, $t' \leftarrow 84, 79, 66, 69, 79, 82, 78, 79, 84, 256, 258$ etc.
- Au final
  $84; 79; 66; 69; 79; 82; 78; 79; 84; 256; 258; 260; 265; 259; 261; 263$

Exemple (Wikipedia)

On veut compresser "**TOBEORNOTTOBEORTOBEORNOT**".

- Position 15 : **T,TO,TOB** sont des clés mais pas **TOBE**.
  $d[\textbf{TOBE}] \leftarrow 268$,
  $t' \leftarrow 84, 79, 66, 69, 79, 78, 79, 84, 84, 256, 258, 260, 265$

Exemple (Wikipedia)

On veut compresser "**TOBEORNOTTOBEORTOBEORNOT**".

- Position 15 : **T,TO,TOB** sont des clés mais pas **TOBE**.
  $d[\textbf{TOBE}] \leftarrow 268$,
  $t' \leftarrow 84, 79, 66, 69, 79, 78, 79, 84, 84, 256, 258, 260, 265$
- Position 18 : **E,EO** sont des clés mais pas **EOR**. $d[\textbf{EOR}] \leftarrow 269$,
  $t' \leftarrow 84, 79, 66, 69, 79, 78, 79, 84, 84, 256, 258, 260, 265, 259$

Exemple (Wikipedia)

On veut compresser "**TOBEORNOTTOBEORTOBEORNOT**".

- Position 15 : **T,TO,TOB** sont des clés mais pas **TOBE**.
  $d[\textbf{TOBE}] \leftarrow 268$,
  $t' \leftarrow 84, 79, 66, 69, 79, 78, 79, 84, 84, 256, 258, 260, 265$
- Position 18 : **E,EO** sont des clés mais pas **EOR**. $d[\textbf{EOR}] \leftarrow 269$,
  $t' \leftarrow 84, 79, 66, 69, 79, 78, 79, 84, 84, 256, 258, 260, 265, 259$
- Position 20 : **R,RN** sont des clés mais pas **RNO**. $d[\textbf{RNO}] \leftarrow 270$,
  $t' \leftarrow 84, 79, 66, 69, 79, 78, 79, 84, 84, 256, 258, 260, 265, 259, 261$

Exemple (Wikipedia)

On veut compresser "**TOBEORNOTTOBEORTOBEORNOT**".

- Position 15 : **T,TO,TOB** sont des clés mais pas **TOBE**.
  $d[\textbf{TOBE}] \leftarrow 268$,
  $t' \leftarrow 84, 79, 66, 69, 79, 78, 79, 84, 84, 256, 258, 260, 265$
- Position 18 : **E,EO** sont des clés mais pas **EOR**. $d[\textbf{EOR}] \leftarrow 269$,
  $t' \leftarrow 84, 79, 66, 69, 79, 78, 79, 84, 84, 256, 258, 260, 265, 259$
- Position 20 : **R,RN** sont des clés mais pas **RNO**. $d[\textbf{RNO}] \leftarrow 270$,
  $t' \leftarrow 84, 79, 66, 69, 79, 78, 79, 84, 84, 256, 258, 260, 265, 259, 261$
- Position 22 à fin : **O,OT** sont des clés.
  $t' \leftarrow 84, 79, 66, 69, 79, 78, 79, 84, 84, 256, 258, 260, 265, 259, 261, 263$

Initialisation

- On note d le dictionnaire (code,facteur) qui est l'inverse de celui de la partie précédente (en fait, puisque l'ensemble des codes forme un intervalle de nombres, un simple tableau redimensionnable suffit).

Initialisation

- On note d le dictionnaire (code,facteur) qui est l'inverse de celui de la partie précédente (en fait, puisque l'ensemble des codes forme un intervalle de nombres, un simple tableau redimensionnable suffit).
- Ce dictionnaire est initialisé ainsi : à tous les codes entre (par exemple 0 et 256) on associe la lettre correspondante de l'alphabet.

Initialisation

- On note d le dictionnaire (code,facteur) qui est l'inverse de celui de la partie précédente (en fait, puisque l'ensemble des codes forme un intervalle de nombres, un simple tableau redimensionnable suffit).
- Ce dictionnaire est initialisé ainsi : à tous les codes entre (par exemple 0 et 256) on associe la lettre correspondante de l'alphabet.
- La notation $|d|$ désigne le nombre d'associations déjà entrées. Avec le code ASCII, $|d| = 256$ au départ.

Initialisation

- On note d le dictionnaire (code,facteur) qui est l'inverse de celui de la partie précédente (en fait, puisque l'ensemble des codes forme un intervalle de nombres, un simple tableau redimensionnable suffit).
- Ce dictionnaire est initialisé ainsi : à tous les codes entre (par exemple 0 et 256) on associe la lettre correspondante de l'alphabet.
- La notation $|d|$ désigne le nombre d'associations déjà entrées. Avec le code ASCII, $|d| = 256$ au départ.
- Le premier code c lu est nécessairement celui d'un unique caractère. On écrit donc $d[c]$ dans le fichier de sortie et on *garde* c en mémoire.

Déroulement

Cas facile : on lit un code connu

On garde en mémoire le précédent code lu c. On lit un code n où $n < |d|$ (ce qui signifie qu'on sait ce que code n) :

- Posons $d[n] = xm'$; x est un caractère et m' un mot.

Déroulement

Cas facile : on lit un code connu

On garde en mémoire le précédent code lu c. On lit un code n où $n < |d|$ (ce qui signifie qu'on sait ce que code n) :

- Posons $d[n] = xm'$; x est un caractère et m' un mot.
- On écrit xm' dans le fichier de sortie

Déroulement

Cas facile : on lit un code connu

On garde en mémoire le précédent code lu c. On lit un code n où $n < |d|$ (ce qui signifie qu'on sait ce que code n) :

- Posons $d[n] = xm'$; x est un caractère et m' un mot.
- On écrit xm' dans le fichier de sortie
- On rajoute ensuite un nouvel élément mx dans le dictionnaire où $m = d[c]$. On pose donc $d[|d|] = mx$.

Déroulement

Cas facile : Pourquoi faut-il ajouter mx à d ?

On reproduit en fait le processus de compression mais en remplissant le dictionnaire avec un temps de retard.

- Il est clair qu'il faut ajouter xm' au texte décompressé quand on lit n dans le texte compressé.

Déroulement

Cas facile : Pourquoi faut-il ajouter mx à d ?

On reproduit en fait le processus de compression mais en remplissant le dictionnaire avec un temps de retard.

- Il est clair qu'il faut ajouter xm' au texte décompressé quand on lit n dans le texte compressé.
- Selon le principe de compression :

Déroulement

Cas facile : Pourquoi faut-il ajouter mx à d ?

On reproduit en fait le processus de compression mais en remplissant le dictionnaire avec un temps de retard.

- Il est clair qu'il faut ajouter xm' au texte décompressé quand on lit n dans le texte compressé.
- Selon le principe de compression :
  - **Dans la compression** : on ajoute une entrée au dictionnaire pour mx quand : 1) on lit x 2) m a un code c connu et 3) $mx \notin d$. Le code de mx devient le plus grand code rencontré jusqu'ici.

Déroulement

Cas facile : Pourquoi faut-il ajouter mx à d ?

On reproduit en fait le processus de compression mais en remplissant le dictionnaire avec un temps de retard.

- Il est clair qu'il faut ajouter xm' au texte décompressé quand on lit n dans le texte compressé.
- Selon le principe de compression :
  - **Dans la compression** : on ajoute une entrée au dictionnaire pour mx quand : 1) on lit x 2) m a un code c connu et 3) $mx \notin d$. Le code de mx devient le plus grand code rencontré jusqu'ici.
  - **Dans la compression** : Le code c de m (qui est connu, sinon on ne serait pas arrivé à x) est ajouté au texte compressé.
    On repart alors avec x comme motif lu.

Déroulement

Cas facile : Pourquoi faut-il ajouter mx à d ?

On reproduit en fait le processus de compression mais en remplissant le dictionnaire avec un temps de retard.

- Il est clair qu'il faut ajouter xm' au texte décompressé quand on lit n dans le texte compressé.
- Selon le principe de compression :
  - **Dans la compression** : on ajoute une entrée au dictionnaire pour mx quand : 1) on lit x 2) m a un code c connu et 3) $mx \notin d$. Le code de mx devient le plus grand code rencontré jusqu'ici.
  - **Dans la compression** : Le code c de m (qui est connu, sinon on ne serait pas arrivé à x) est ajouté au texte compressé.
    On repart alors avec x comme motif lu.
  - **Dans la décompression** : quand on lit le code n de xm', après avoir lu c (code de m) on sait que le code du facteur mx n'a jamais été rencontré et donc qu'il faut l'ajouter. On a ainsi un nouveau code : il se range en position $|d|$, c.a.d juste après le plus grand code rencontré jusqu'ici.

Déroulement

Cas problématique : $n = |d|$

Le code n lu est tel que $n = |d|$, donc on lit un code non encore présent dans la table de décompression.

- On lit le code n : il a été placé à cet endroit au moment de la compression après avoir lu un wy. Ainsi, n est le code de w.

Déroulement

Cas problématique : $n = |d|$

Le code n lu est tel que $n = |d|$, donc on lit un code non encore présent dans la table de décompression.

- On lit le code n : il a été placé à cet endroit au moment de la compression après avoir lu un wy. Ainsi, n est le code de w.
- n est maximal parmi les codes déjà rencontrés. Revenant au moment de la compression, cela signifie que w est le dernier facteur qui a produit un code avant d'écrire n.

Déroulement

Cas problématique : $n = |d|$

Le code n lu est tel que $n = |d|$, donc on lit un code non encore présent dans la table de décompression.

- On lit le code n : il a été placé à cet endroit au moment de la compression après avoir lu un wy. Ainsi, n est le code de w.
- n est maximal parmi les codes déjà rencontrés. Revenant au moment de la compression, cela signifie que w est le dernier facteur qui a produit un code avant d'écrire n.
- Or, juste avant n dans le texte compressé, il y a c (lequel code m). Ainsi w est de la forme mx.

Déroulement

Cas problématique : $n = |d|$

Le code n lu est tel que $n = |d|$, donc on lit un code non encore présent dans la table de décompression.

- On lit le code n : il a été placé à cet endroit au moment de la compression après avoir lu un wy. Ainsi, n est le code de w.
- n est maximal parmi les codes déjà rencontrés. Revenant au moment de la compression, cela signifie que w est le dernier facteur qui a produit un code avant d'écrire n.
- Or, juste avant n dans le texte compressé, il y a c (lequel code m). Ainsi w est de la forme mx.
- Dans la compression, après avoir lu $w = mx$, on repart de x et on lit wy, c.a.d. mxy. Ainsi, la 1ere lettre de m est x !

Déroulement

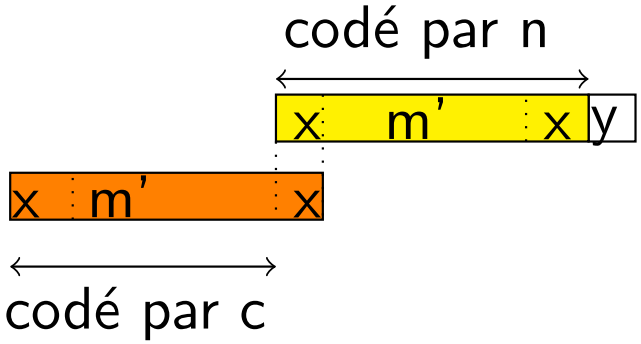
Cas problématique : $n = |d|$

Le code n lu est tel que $n = |d|$, donc on lit un code non encore présent dans la table de décompression.

- On lit le code n : il a été placé à cet endroit au moment de la compression après avoir lu un wy. Ainsi, n est le code de w.
- n est maximal parmi les codes déjà rencontrés. Revenant au moment de la compression, cela signifie que w est le dernier facteur qui a produit un code avant d'écrire n.
- Or, juste avant n dans le texte compressé, il y a c (lequel code m). Ainsi w est de la forme mx.
- Dans la compression, après avoir lu $w = mx$, on repart de x et on lit wy, c.a.d. mxy. Ainsi, la 1ere lettre de m est x !
- On ajoute mx au texte décompressé et on réalise l'association $d[n] = mx$.

Cas problématique : $n = |d|$

n code un wy tel que le code de w est le dernier lu. En posant $m = xm'$, on a $w = xm'x$.



Algorithme

On initialise le dictionnaire avec l'alphabet (par exemple alphabet ASCII des caractères codés sur 8 bits). La fonction **Lire** lit le code courant de T' et positionne le curseur sur le code suivant.

Listing 2 – Décompression

```
1  fonction lzw_decompress(T' : texte compressé,
2                          d : dictionnaire (code, facteur)):
3      c ← Lire(T'); /*1er code lu*/
4      /*le 1er code correspond toujours à une lettre*/
5      Ecrire(d[c]); /*ajouter le texte codé par c*/
6      tant_que il reste un code non lu de T' faire
7          n ← Lire(T'); /*code courant*/
8          si n est une clef de d /*code n déjà rencontré*/
9              alors e ← d[n]; /*décompression*/
10                 d[|d|] ← d[c] · e[0] /*nouvelle association*/
11             sinon /*décompression, cas n = |d|*/
12                 e ← d[c] · d[c][0];
13                 d[|d|] ← e /*nouvelle association*/
14         Ecrire(e);
15         c ← n
16     fin_faire
```

Taille des entiers en OCaml

En OCaml, les entiers sont un bit plus court que les entiers machines. Sur la plupart des machines, les entiers sont de taille 32 ou 64 bits. En OCaml, les entiers sont donc de taille 31 ou 63 bits.
Or, le premier bit est un bit de signe, les entiers positifs sont donc codés entre 0 et $2^{30} - 1$ (ou $2^{62} - 1$).

La représentation du résultat de la compression par une liste d'entiers OCAML n'est pas très réaliste : il faudrait *a priori* 30 bits (ou 62) pour stocker chaque entier. Cependant, on remarque que la taille des entiers produits par l'algorithme de compression croît progressivement au fur et à mesure que l'on avance dans la liste (et que le dictionnaire se remplit). Dans la pratique, on peut donc utiliser la technique suivante pour coder la liste :

- Tant que tous les entiers sont strictement inférieurs à 255, coder ces entiers sur 8 bits.

La représentation du résultat de la compression par une liste d'entiers OCAML n'est pas très réaliste : il faudrait *a priori* 30 bits (ou 62) pour stocker chaque entier. Cependant, on remarque que la taille des entiers produits par l'algorithme de compression croît progressivement au fur et à mesure que l'on avance dans la liste (et que le dictionnaire se remplit). Dans la pratique, on peut donc utiliser la technique suivante pour coder la liste :

- Tant que tous les entiers sont strictement inférieurs à 255, coder ces entiers sur 8 bits.
- Lorsque l'on rencontre le premier entier supérieur ou égal à 255, émettre la séquence 11111111 (huit fois le bit 1) et continuer, tant que les entiers sont strictement inférieurs à 511, en codant les entiers sur 9 bits.

La représentation du résultat de la compression par une liste d'entiers OCAML n'est pas très réaliste : il faudrait *a priori* 30 bits (ou 62) pour stocker chaque entier. Cependant, on remarque que la taille des entiers produits par l'algorithme de compression croît progressivement au fur et à mesure que l'on avance dans la liste (et que le dictionnaire se remplit). Dans la pratique, on peut donc utiliser la technique suivante pour coder la liste :

- Tant que tous les entiers sont strictement inférieurs à 255, coder ces entiers sur 8 bits.
- Lorsque l'on rencontre le premier entier supérieur ou égal à 255, émettre la séquence 11111111 (huit fois le bit 1) et continuer, tant que les entiers sont strictement inférieurs à 511, en codant les entiers sur 9 bits.
- Lorsque l'on rencontre le premier entier supérieur ou égal à 511, émettre la séquence 111111111 (neuf fois le bit 1) et continuer, tant que les entiers sont strictement inférieurs à 1023, en codant les entiers sur 10 bits.

- De manière générale, tant que les entiers considérés sont strictement inférieurs à $n = 2^k - 1$, on peut les représenter sur k bits.

- De manière générale, tant que les entiers considérés sont strictement inférieurs à $n = 2^k - 1$, on peut les représenter sur k bits.
- Lorsque le premier entier supérieur ou égal à $2^k - 1$ est rencontré, on émet la séquence $1 \ldots 1$ (k fois le bit 1) et on continue en codant les entiers sur $k + 1$ bits.

- De manière générale, tant que les entiers considérés sont strictement inférieurs à $n = 2^k - 1$, on peut les représenter sur k bits.
- Lorsque le premier entier supérieur ou égal à $2^k - 1$ est rencontré, on émet la séquence $1 \ldots 1$ (k fois le bit 1) et on continue en codant les entiers sur $k + 1$ bits.
- Donc si le code 10 se trouve au début de la liste des codages, il prend 8 bits d'espace mais après le premier nombre plus grand que 255, il prend 9 bits etc.