Chapitre 15

Algorithmique du texte

Notions	Commentaires
Recherche dans un texte. Algorithme de Boyer-Moore. Algorithme de Rabin-Karp.	On peut se restreindre à une version simplifiée de l'algorithme de Boyer-Moore, avec une seule fonction de décalage. L'étude précise de la complexité de ces algorithmes n'est pas exigible.
Compression. Algorithme de Huffman. Algorithme Lempel-Ziv-Welch.	On explicite les méthodes de décompression associées.

Extrait de la section 4.4 du programme officiel de MP2I : « Algorithmique des textes ».

Sommaire

0 Généralités

Définition 1.

- Un **alphabet** Σ est un ensemble fini non vide dont les éléments sont appelés **lettres** (ou **symboles** ou **caractères**).
- Un **mot** (ou **chaîne de caractères**) m sur Σ est

- L'ensemble des mots sur Σ est noté Σ^*.

Remarque.

- ⚠ S'il figure parmi les symboles de notre alphabet, l'espace est un caractère comme les autres. Ainsi, un "mot" comme nous les avons définis peut tout à fait contenir un ou plusieurs espaces, et être très long.
- On confond souvent une lettre c et le mot constitué d'une unique lettre c, mais ce sont bien deux objets différents. Par exemple, on peut penser aux types `char` et `string` en OCaml.
- On peut voir le mot vide ϵ comme $\epsilon = m_1 \dots m_n$ avec $n = 0$.
- Donner une définition inductive des mots :

Exemples : Alphabets courants (en informatique).

- $\Sigma = \{a, b\}$ ou $\Sigma = \{0, 1\}$ (flux de bits).
- alphabet "usuel" à 26 lettres
- ensemble des 256 (ou plutôt 128) caractères ASCII standards
- $\Sigma = \{A, T, G, C\}$ pour les séquences d'ADN

Définition 2.

- La **longueur** d'un mot $m \in \Sigma^*$, notée $|m|$, est définie par :
  - $|\epsilon = 0|$
  - $|m_1 \dots m_n| = n$ (où $m_1, \dots, m_n \in \Sigma$)
- L'ensemble des mots de longueur n sur un alphabet Σ est noté Σ^n.
- Le **nombre d'occurrences** d'une lettre c dans m, ou longueur en c de m, notée $|m|_c$, est définie par :

$$|m|_c = \text{Card}\{i \in [\![1; n]\!] \mid m_i = c\}$$

Définition 3.

- La **concaténation** de deux mots u et v sur un même alphabet Σ, notée $u \cdot v$, est définie par :
  - $\epsilon \cdot v = v$
  - $u \cdot \epsilon = u$
  - $(u_1 \dots u_n) \cdot (v_1 \dots v_p) = u_1 \dots u_n v_1 \dots v_p$
- Pour un mot u et un entier $n \geqslant 0$, on définit u^n par :
  - $u^0 = \epsilon$
  - $u^{n+1} = u \cdot u^n$

Remarque.

On écrit souvent juste uv à la place $u \cdot v$. ⚠ Il y a alors une ambiguïté entre concaténation et suite de lettres. À n'utiliser que lorsque le contexte est clair.

Proposition 4.

- La concaténation est associative : on écrit $u \cdot v \cdot w$ pour $u \cdot (v \cdot w) = (u \cdot v) \cdot w$.
- Les règles de calcul usuelles sur les puissances s'appliquent :

$$u^m \cdot u^n = u^{m+1}$$

$$(u^m)^n = u^{mn}$$

Démonstration. Propriétés immédiates. Laissées en exercice. □

Définition 5.

Soient $u, v \in Sigma^*$ deux mots sur un même alphabet Σ. On dit que :

- u est un **préfixe** de v s'il existe un mot w tel que $v = u \cdot w$.
- u est un **suffixe** de v s'il existe un mot w tel que $v = w \cdot u$.
- u est un **facteur** de v s'il existe deux mots w et z tels que $v = w \cdot u \cdot z$.

On dit que u est un préfixe (respectivement suffixe ou facteur) **strict** de v si on a de plus $u \neq v$.

Exemple. Sur l'alphabet usuel :

- le mot *or* est un préfixe du mot *ornement*.
- le mot *ment* est un préfixe du mot *ornement*.
- le mot *nem* est un facteur du mot *ornement*.

Remarque.

- u et ϵ sont toujours des préfixes, suffixes et facteurs du mot u.
- Un préfixe ou suffixe de u est un facteur de u.

Exercice : Considérons $u = abab$. Donner tous les préfixes, suffixes et facteurs de u.

Proposition 6.
Soient $u, u', v, v', w \in \Sigma^*$.

-

- Plus généralement, si $u \cdot v = u' \cdot v'$, alors il existe un unique mot t tel que :
  - soit $u = u' \cdot t$ et $v' = t \cdot v$;
  - soit $u' = u \cdot t$ et $v = t \cdot v'$.

Démonstration. •

- Laissé en exercice. (Ce n'est pas difficile, mais un peu pénible à écrire.)

□

Remarque.

- Si u est un préfixe de u' et u' est un préfixe de u alors $u = u'$. De même pour les suffixes.
- « Être préfixe (resp. suffixe) de » est une :

Définition 7.
Soient $u, v \in \Sigma^*$. Notons $u = u_1 \dots u_n$ (avec $n = 0$ si u est vide).
On dit que u est un **sous-mot** de v s'il existe des mots $t_0, t_1, \dots t_n$ tels que $v = t_0 \cdot u_1 \cdot t_1 \cdot u_2 \cdot \dots \cdot t_{n-1} \cdot u_n \cdot t_n$.
Autrement dit, u est un sous-mot de v si u est une suite extraite (de lettres) de v, i.e. s'il existe une fonction d'extraction $\varphi : [\![1; |u|]\!] \to [\![1; |v|]\!]$ strictement croissante telle que $u_i = v_{\varphi(i)} \quad \forall i \in [\![1; |u|]\!]$.

Remarque.

- u et ϵ sont toujours des sous-mots de u.
- ⚠ Ne pas confondre sous-mot et facteur ! Quel lien y a-t-il entre les deux ?

Exemple. Les sous-mots de *abcb* sont :

1 Compressions et décompressions

1.0 Principe général

Objectif : Réduire l'espace occupé par une information (ex : document à télécharger).

Définition 8.
Soient Σ, Σ' deux alphabets.
Un **processus de compression sans perte** d'un texte (mot) sur Σ (au sens large) est la donnée d'un couple de fonctions $(comp, decomp)$ avec :
$\begin{cases} comp : \Sigma^* \to \Sigma'^* \\ decomp : \Sigma'^* \to \Sigma^* \end{cases}$
telles que :

- $decomp \circ comp = \mathrm{Id}_{\Sigma^*}$
- pour la plupart des mots m qui nous intéressent, $|comp(m)| < |m|$.

Remarque.

- Il faut que $comp$ soit injective, mais on n'impose pas la surjectivité (et ce n'est souvent pas le cas). On n'a pas en général $comp \circ decomp = \mathrm{Id}_{\Sigma'^*}$, et $decomp$ n'est généralement définie que sur $comp(\Sigma^*)$.
- Il est impossible de compresser efficacement toutes les entrées possibles (pensez par exemple qu'il n'existe pas d'injection de Σ^n dans Σ^p avec $p < n$). Si on travaille en binaire par exemple ($\Sigma = \Sigma' = \{0, 1\}$), on aura toujours des mots tels que $|comp(m)| \geqslant |m|$. On veut que $|comp(m)| < |m|$ sur des entrées typiques, fréquentes. Par exemple :
  - Les mots d'une langue (Français, etc.)
  - du code exécutable
  - un ensemble de triangles représentant la géométrie d'une scène

Démonstration d'une compression de la vie courante :

Les deux commandes suivantes créent chacune un fichier de 10 mégaoctets :

- `random.bin` constitué d'octets (pseudo)-aléatoires
- `zero.bin` constitué intégralement de zéros.

Terminal
```
$ dd if=/dev/urandom of=random.bin bs=10MB
  ↪ count=1

1+0 records in
1+0 records out
10000000 bytes (10 MB, 9,5 MiB) copied,
  ↪ 0,0327111 s, 306 MB/s
```

Terminal
```
$ dd if=/dev/zero of=zero.bin bs=10MB
  ↪ count=1

1+0 records in
1+0 records out
10000000 bytes (10 MB,  9,5 MiB) copied,
  ↪ 0,00807603 s, 1,2 GB/s
```

Compression de ces deux fichiers via l'outil `zip` :

Terminal
```
$ zip -r random.zip random.bin
adding: random.bin (deflated 0%)
```

Terminal
```
$ zip -r zero.zip zero.bin
adding: zero.bin (deflated 100%)
```

`zip` nous indique qu'il a réussi à économiser 0% de place dans le cas de `random.bin` et 100% dans le cas de `zero.bin`. Bien évidemment, il s'agit de valeurs approchées. On peut récupérer les tailles exactes (en octets) :

Terminal

```
$ du -b *.zip
10001700        random.zip
9885        zero.zip
$
```

On voit donc que la taille du fichier constitué de zéros a été divisée par plus de mille, alors que celle du fichier aléatoire a légèrement augmenté après compression.

Définition 9.
Compression avec perte (culture G.) :
On n'impose plus que $decomp(comp(m)) = m$ mais qu'il "ressemble" à m. On peut par exemple définir cette ressemblance par une distance mathématique dans un certain espace. Mais ce qui nous intéresse en général est simplement qu'un humain ne voie pas trop la différence entre le fichier décompressé et le fichier original (avant compression).

Exemple. compression de fichiers MP3 (son), jpg (image) ou de codecs video.

1.1 Code binaire

Dans cette section et la suivante, on considère un texte (mot) de N caractères à compresser, écrit sur un alphabet Σ.

Définition 10.
Un **code binaire** sur Σ est une application $f : \Sigma \to \{0,1\}^* \backslash \{\epsilon\}$ injective.
À chaque lettre de Σ, on associe une suite finie (non vide) de 0 et de 1.

Définition 11.
On étend un code binaire f en une fonction sur les mots notée $\bar{f}$ (aussi souvent notée f pour alléger), de la manière suivante :

$$\bar{f} : \begin{cases} \Sigma^* \to \{0,1\}^* \\ \epsilon \mapsto \epsilon \\ a_1 \dots a_n \mapsto f(a_1) \cdot \ \dots \ \cdot f(a_n) \end{cases}$$

Le codage d'un mot est la concaténation des codages des caractères du mot.

Exemple. (Rappel) Comment sont représentées les caractères ASCII en machine ? En déduire un code binaire des chaînes de caractères.

Sur combien de bits la chaîne de caractères ASCII ”magicienne” est-elle représentée en machine ?

Définition 12.
Un code binaire est dit **à longueur fixe** si tous les $f(c)$ pour $c \in \Sigma$ sont de même longueur.
Sinon, il est dit **à longueur variable**.

Exemple. La représentation ASCII fournit un code à longueur fixe.

Exercice :

1. Considérons l'alphabet $\Sigma = \{a, c, e, g, i, n, m\}$ à 7 lettres.
   Encodez le mot ”magicienne” par un code binaire à longueur fixe aussi efficacement que possible (i.e. en utilisant le moins de bits possible). Combien de bits avez-vous utilisé ?
2. Proposer un code aussi efficace que possible pour encoder un mot à N caractères écrit sur un alphabet Σ_b à b lettres. Combien de bits utilise-t-il ?

Définition 13.
Un code binaire f est dit **uniquement déchiffrable** si son extension $\bar{f}$ est injective. Sinon, il est dit **ambigu**.

Exemple.

Considérons l'alphabet $\Sigma = \{a, b, c\}$ et le code binaire $f : \begin{cases} a \mapsto 01 \\ b \mapsto 010 \\ c \mapsto 1 \end{cases}$

f est-il uniquement déchiffrable ou ambigu ?

Définition 14.
Un code binaire f est dit **préfixe**[i] s'il n'existe pas de couple $(a, b) \in \Sigma^2$ tel que $a \neq b$ et $f(a)$ est un préfixe de $f(b)$.

i. ou **sans-préfixe**, en Anglais *prefix-free*, ce qui est plus logique

Exemple. Le code de l'exemple précédent est-il préfixe ?

Proposition 15.
Tout code préfixe est uniquement déchiffrable.

Démonstration.

□

Définition 16.
Le **poids** d'un code binaire f sur un mot $m \in \Sigma^*$, noté $w_m(f)$, est la longueur totale de l'image du mot m par f :

$$\boxed{w_m(f) = |f(m)| = \sum_{c \in \Sigma} |f(c)|.|m|_c}$$

où on rappelle que : $\begin{cases} |f(c)| \text{ est la longueur du code associé au caractère } c \text{ par la fonction } f \\ |m|_c \text{ est le nombre d'occurrences de la lettre } c \text{ dans le mot } m \end{cases}$

Remarque. $w_m(f)$ est la longueur du texte compressé m par le code binaire f. On cherche à construire un code uniquement déchiffrable qui minimise cette quantité. On va chercher à construire un code binaire préfixe, afin de s'assurer qu'il soit uniquement déchiffrable.

1.2 Algorithme de Huffman

Idée générale :

Exemple. Pour encoder le mot ”magicienne”, on voit que les lettres ”i”, ”e” et ”n” apparaissent plusieurs fois. Pour leur donner un code plus court, proposons le code binaire suivant :

$$f : \begin{cases} a \mapsto 0000 \\ c \mapsto 0001 \\ e \mapsto 10 \\ g \mapsto 0010 \\ i \mapsto 11 \\ n \mapsto 01 \\ m \mapsto 0011 \end{cases}$$

Quel est l'encodage du mot ”magicienne” avec ce code binaire ? Combien de bits utilise-t-il ? Comparer avec les encodages trouvés précédemment dans ce cours.

Remarque. ⚠ Les espaces ne font pas partie du texte compressé ! On a une simple suite de bits.

Définition 17.
L'**arbre binaire associé** à un code binaire f préfixe est l'unique arbre tel que :

- une arête vers un fils gauche représente un bit 0
- une arête vers un fils droit représente un bit 1
- les noeuds internes ne sont pas étiquetés
- les feuilles sont étiquetées par les caractères de l'alphabet. Une feuille porte l'étiquette $c \in \Sigma$ si et seulement si le chemin de la racine vers le cette feuille donne le code binaire du caractère c : $f(c)$.

Exemple. Dessiner l'arbre binaire associé au code f précédent.

Compression d'un texte :

Pour le moment, supposons qu'on dispose d'un codage préfixe optimal f (et donc de son arbre associé A).

Expliquer comment compresser un texte :

Exemple.

Sur le mot magicienne, on a obtenu le code "11000001110000110001010110101 0".

Complexité :

- Complexité naïve (en utilisant l'arbre) : On doit faire un parcours entier de l'arbre du code pour trouver chaque chemin de la racine vers une feuille jusqu'à trouver le caractère qu'on veut coder. On obtient du $O(n|A|)$.
- On peut utiliser un dictionnaire pour retenir ces chemins. Dans l'entrée c du dictionnaire, on stocke son code binaire $f(c)$. Si on accède en temps constant aux clés du dictionnaire, la complexité totale est réduite à $O(|C|)$ avec $|C| \leqslant N.h_A$ la longueur du code compressé par le code f, et h_A la hauteur de l'arbre A.

Décompression d'un texte :

Expliquer comment décompresser le code d'un texte compressé :

Exemple.
Décoder le texte compressé suivant, obtenu avec le même arbre de code que le mot "magicienne" précédent : 000111001110.

Trouver un codage préfixe optimal :

Principe : L'**algorithme de Huffman** (aussi appelé codage de Huffman) consiste à trouver un code binaire préfixe optimal pour un mot donné à compresser. Expliquer comment.

Remarque.

- On considère indifféremment les fréquences d'apparition de chaque lettre ou les nombres d'occurrence de chaque lettre dans le mot (ce sont les mêmes à un facteur multiplicatif près).
- Il n'y a pas un unique code optimal possible. On ne parlera donc jamais du code binaire optimal mais d'un code binaire optimal.
- Il faut que l'alphabet contienne au moins deux lettres, pour que les lettres soient encodées par au moins un bit chacune. On supposera dans toute la suite que c'est bien le cas, le cas d'un alphabet unaire étant un peu particulier.

Exemple.
Sur le texte à compresser "magicienne", on a les nombres d'occurrences suivants de chaque lettre :

a	c	e	g	i	n	m
1	1	2	1	2	2	1

Dérouler l'algorithme de Huffman pour construire le code binaire :

Exercice : Trouver un codage optimal pour le mot "élément". *Partir d'un alphabet bien choisi et ignorer les accents.*

Théorème 18.
Le codage de Huffman h est optimal, c'est à dire qu'il minimise le poids $w_m(h)$ du code binaire h sur le mot donné m.

Proposition 19.
Lemme : L'arbre associé à un code optimal est binaire strict.

Démonstration. □

Démonstration.

Commençons par **formaliser** un peu le problème. L'optimalité d'un code ne dépend que des lettres qui composent le mot à compresser et de leurs nombres d'occurrences dans le mot, pas du mot lui-même (peu importe l'ordre d'apparition des lettres par exemple). Notons $\Sigma_m = \{(a_1, n_1), \dots (a_p, n_p)\}$ l'alphabet Σ enrichi des nombres d'occurrences de chaque lettre dans le mot m. On énumère les lettres dans l'ordre croissant de nombres d'occurrences : $n_1 \leqslant n_2 \leqslant \cdots \leqslant n_p$.

On note $opt(\Sigma_m)$ le poids minimal d'un code pour le mot m (sur l'alphabet Σ_m). On veut montrer que :

$w_m(h) = opt(\Sigma_m)$ où h est le code binaire produit par l'algorithme de Huffman.

- **(propriété d'échange :)** Tout d'abord, montrons qu'il existe un code optimal pour Σ_m dans lequel la feuille étiquetée a_1 a pour soeur la feuille étiquetée a_2 [1] :

- **(lemme pour la récurrence :)** Si $|\Sigma_m \geqslant 3$, on définit $|\Sigma'_m| = \{(b, n_1+n_2), (a_3, n_3), \dots, (a_p, n_p)\}$ où b est une nouvelle lettre, distincte de toutes les autres. Alors montrons qu'on a : $\boxed{opt(\Sigma_m) \geqslant opt(\Sigma'_m) + n_1 + n_2}$.

1. le premier choix glouton étant de les regrouper dans l'arbre de code construit, on montre bien qu'il existe une solution optimale qui fait le premier choix glouton

- **(conclusion par récurrence :)** Montrons que le codage de Huffman est optimal.

□

Exemple.

Voici ci-dessous l'arbre de Huffman obtenu sur le texte intégral du roman *Le Tour du monde en quatre-vingt jours* de Jules Verne. Il y a 428 775 caractères au total dans ce texte, pour 81 caractères différents (les accents ont été supprimés pour limiter le nombre de caractères). Certains caractères apparaissent souvent dans le texte, comme 'e' (51 955 occurrences) et '␣' (i.e. espace, 67 104 occurrences).

Exercice :
Donner le code associé à la lettre 'e' :

Donner le code associé à la lettre 'Z' :

Quelle est la lettre encodée par la séquence '110' ?

Quelle est la lettre encodée par la séquence '10100011001011' ?

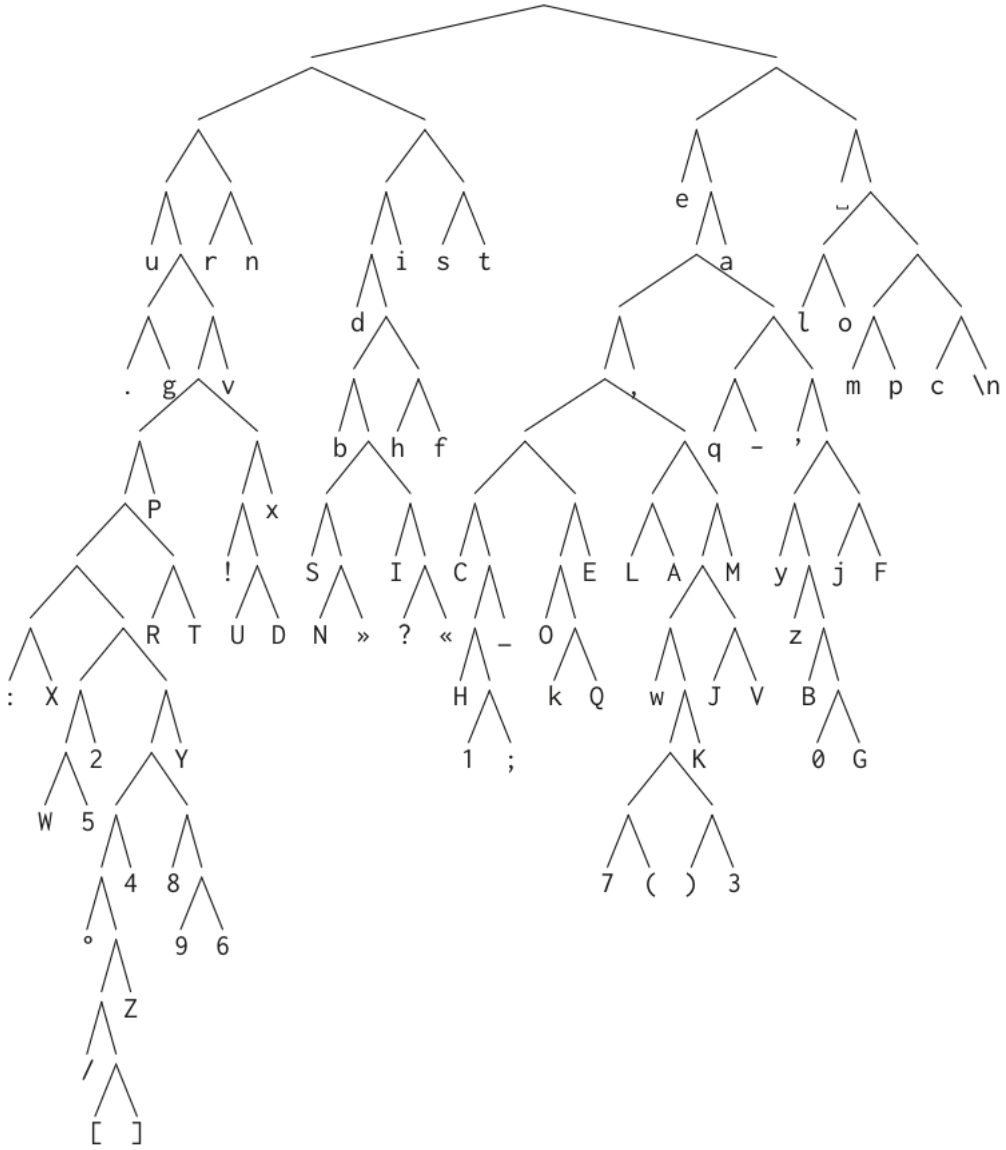

FIGURE 15.1 – Arbre de Huffman pour *Le Tour du monde en quatre-vingt jours*
Image issue de *Informatique-MP2I/MPI*, de Balabonski et al.

Efficacité :

Au final, le texte compressé occupe 1 919 473 bits, contre 3 430 200 au départ (428 775 caractères × 8 bits par caractère, en supposant un encodage 8 bits type Latin-1), soit une économie de 44%. Ce qui signifie que l'espace occupé après compression n'est que 56% de ce qu'il était avant compression.

Remarque.

- Le codage de Huffman nécessite de transmettre l'arbre du code en plus du texte compressé. Il faut le prendre en compte dans l'espace occupé par le texte compressé.
- Le codage de Huffman nécessite un pré-traitement sur le texte entier, une première lecture pour trouver les fréquences et ainsi construire l'arbre. En particulier, on ne peut pas compresser un flux de données à la volée.

Variantes possibles :

- Plutôt que de construire l'arbre sur les fréquences d'apparition dans le texte entier, on peut se limiter à une analyse statistique sur une petite portion du texte.
- On peut aussi se contenter d'utiliser directement les fréquences moyennes d'apparition des lettres dans la langue française, par exemple, pour se passer du prétraitement.

1.3 Algorithme de Lempel-Ziv-Welch

Idée générale :

Remarque. On fait le tout en une seule passe, sans préconstruire les motifs en amont. On peut donc recevoir le texte à compresser au fur et à mesure, par exemple s'il est trop gros pour être lu intégralement avant la compression.

Compression d'un texte :

Algorithme :

1) On commence avec une association lettre → code pour chaque lettre de l'alphabet. On "retient" initialement chaque motif constitué d'une unique lettre.
2) On lit le texte de gauche à droite et on construit sa compression à la volée. À chaque étape, on extrait du texte autant de lettres que possible de sorte que le préfixe qu'on lit reste un motif retenu. La première lettre qu'on lit en est forcément un (les lettres sont dans les motifs initiaux).

   Puis, on ajoute le code associé au motif trouvé à la compression du texte en cours de construction, et on retient un motif supplémentaire constitué de celui qu'on vient de lire auquel on ajoute la lettre suivante dans le mot. On lui associe un nouveau code (pas encore utilisé).

Exemple. Sur l'alphabet $\{E, D, N, T\}$, compressons le mot "ENTENDENT" par l'algorithme LZW, pas à pas.

étape	associations motif → code	texte à lire	résultat de la compression
début	E ↦ 0 ; D ↦ 1 ; N ↦ 2 ; T ↦ 3	ENTENDENT	
	E ↦ 0 ; D ↦ 1 ; N ↦ 2 ; T ↦ 3		
	E ↦ 0 ; D ↦ 1 ; N ↦ 2 ; T ↦ 3		
(*)	E ↦ 0 ; D ↦ 1 ; N ↦ 2 ; T ↦ 3		
(**)	E ↦ 0 ; D ↦ 1 ; N ↦ 2 ; T ↦ 3		
	E ↦ 0 ; D ↦ 1 ; N ↦ 2 ; T ↦ 3		
	E ↦ 0 ; D ↦ 1 ; N ↦ 2 ; T ↦ 3		
	E ↦ 0 ; D ↦ 1 ; N ↦ 2 ; T ↦ 3		

Légende :

Explications :
Étape (*) à (**) :

Pour retenir les associations motif → code, on peut utiliser un dictionnaire, dont les clés sont les motifs et les valeurs sont leurs codes associés.

Exercice :
Sur l'alphabet $\{A, L, P, R\}$, compresser le mot "RAPLAPLA" par la méthode LZW.

Décompression :

On pourrait transmettre le dictionnaire obtenu pendant la compression, de la même manière qu'on transmet l'arbre de Huffman, mais on n'en a pas besoin pour décompresser le code ! On peut reconstruire le dictionnaire au fur et à mesure de la décompression.

Algorithme :

1) On démarre avec les mêmes associations initiales, renversées (code → lettre).
2) À chaque étape, on lit un code et on le décompresse en donnant le motif associé à ce code dans le dictionnaire. À partir du 2ème motif lu, on peut déduire quel nouveau motif a été ajouté dans le dictionnaire à la précédente étape de compression. En effet, on connait maintenant la lettre qui a suivi ce précédent motif dans le mot (la première du motif qu'on vient de décompresser). On construit donc le dictionnaire progressivement en ajoutant les associations code → motif à chaque nouveau motif décompressé (avec un motif de retard par rapport à la compression).

Exemple. Décompressons ensemble le code obtenu précédemment pour ”ENTENDENT”, pas à pas :

étape	associations code → motif	code compressé	résultat de la décompression
début	0 ↦ E ; 1 ↦ D ; 2 ↦ N ; 3 ↦ T	0 2 3 4 1 4 3	
	0 ↦ E ; 1 ↦ D ; 2 ↦ N ; 3 ↦ T		
	0 ↦ E ; 1 ↦ D ; 2 ↦ N ; 3 ↦ T		
	0 ↦ E ; 1 ↦ D ; 2 ↦ N ; 3 ↦ T		
	0 ↦ E ; 1 ↦ D ; 2 ↦ N ; 3 ↦ T		
	0 ↦ E ; 1 ↦ D ; 2 ↦ N ; 3 ↦ T		
	0 ↦ E ; 1 ↦ D ; 2 ↦ N ; 3 ↦ T		
	0 ↦ E ; 1 ↦ D ; 2 ↦ N ; 3 ↦ T		

⚠ **Cas délicat possible :** si le code compressé c qu'on lit correspond exactement au dernier motif m ajouté au dictionnaire pendant la compression. Dans ce cas, à cause du décalage (on a un motif de retard sur ce qu'on déduit), on se retrouve à devoir décompresser un code qui n'est pas encore dans le dictionnaire.

Mais on sait que dans ce cas :

- Le motif m auquel ce code c est associé a été ajouté au dictionnaire lors de la lecture et compression du précédent motif m', qui est connu. Donc le motif recherché m est de la forme $m = m'x$ avec x une lettre. Plus précisément, x est la lettre venant après le motif m' dans le mot compressé.
- De plus, le motif m qu'on doit décompresser commence par cette lettre, puisque c'est le motif qui suit m' dans le texte initial. Comme $m = m'x$ et m commence par x, alors x est aussi la première lettre de m' (connu).

On a donc réussi à reconstituer le motif inconnu à décompresser : c'est le motif précédent m' auquel on rajoute sa première lettre à la fin. On peut alors décompresser le code et poursuivre l'algorithme normalement.

Exemple.

- Compresser le mot ”LALALALALERE” par l'algorithme LZW.

- Décompressons ensemble le code obtenu :

étape	associations code → motif	code compressé	résultat de la décompression
début	0 ↦ A ; 1 ↦ E ; 2 ↦ L ; 3 ↦ R	2 0 4 6 5 1 3 1	
	0 ↦ A ; 1 ↦ E ; 2 ↦ L ; 3 ↦ R		
	0 ↦ A ; 1 ↦ E ; 2 ↦ L ; 3 ↦ R		
	0 ↦ A ; 1 ↦ E ; 2 ↦ L ; 3 ↦ R		
	0 ↦ A ; 1 ↦ E ; 2 ↦ L ; 3 ↦ R		
	0 ↦ A ; 1 ↦ E ; 2 ↦ L ; 3 ↦ R		
	0 ↦ A ; 1 ↦ E ; 2 ↦ L ; 3 ↦ R		
	0 ↦ A ; 1 ↦ E ; 2 ↦ L ; 3 ↦ R		

Exercice :
sur l'alphabet $\{A, L, N, P, R, T\}$ de 6 lettres, décompresser les codes suivants :

1. 4 0 3 1 7 9

2. 4 0 2 5 7 3 1 7

Réponses :

Remarque. Pour programmer cet algorithme en pratique, si on veut compresser une chaîne de caractères par exemple, il faut faire des choix d'implémentation. En particulier pour :

- l'alphabet initial : on peut par exemple considérer les texte ASCII (256 caractères), et donc réserver les 256 premiers codes à ces lettres. Le premier motif ajouté aura le code 256 (257 si on commence à 1). On peut aussi travailler uniquement sur l'alphabet initial $\{0, 1\}$, en lisant les représentations binaires des caractères. On voit le texte à compresser comme une suite de bits.
- les codes des motifs (par exemple notre "3 0 2 1 5 7" obtenu) : en général, on fixe la taille sur laquelle on écrit les codes (typiquement $d = 12$ bits). Ceci limite le nombre de codes possibles (à 2^d codes possibles, ici 4096). Quand le dictionnaire est plein, ce qui arrive régulièrement sur des textes un peu long, on peut par exemple arrêter de remplir le dictionnaire, ou décider d'oublier régulièrement les motifs contenus et repartir du dictionnaire initial, etc.

  On peut aussi utiliser des codes de taille variable, qu'on augmente progressivement.

Exemple.

Faisons quelques tests de compression sur *Le Tour du monde en quatre-vingt jours*, de Jules Verne.

- Avec un alphabet $\{0, 1\}$ et des codes de taille variable, on obtient une économie de 29% de bits (contre 44% pour le codage de Huffman, ce qui est moins bien).
- Avec un alphabet à 256 caractères et la taille des codes fixée à $d = 16$ bits, on peut atteindre jusqu'à 59% d'économie.
  - -> Pour $d < 16$, la compression est moins bonne car le dictionnaire contient moins de motifs et identifie moins de répétitions.
  - -> Pour $d > 16$, la compression est aussi moins bonne cette fois car l'écriture des codes prend trop d'espace (ce qui rallonge la compression obtenue).

⚠ Ceci n'est qu'un exemple, d'autres textes pourraient être mieux compressés avec des codes à taille variable, par exemple.

Remarque. Le format `.zip` utilise une sorte de combinaison de codage de Huffman et d'algorithme LZW, appelé DEFLATE.

2 Recherche dans un texte

Incoming.