Algorithmique du texte

I - 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 soit le mot vide (noté ϵ), soit une finie $m_1 \dots m_n$ d'éléments de Σ.
- L'ensemble des mots sur Σ est noté Σ^*.

Remarques :

— ⚠ 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 $n = 0$.

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 1

- 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 :

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

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

Définition 4

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*.

Remarques :

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

Exemple : Considérons $u = abab$. Les préfixes de u sont : ϵ, a, ab, aba, $abab$. Les suffixes de u sont ϵ, b, ab, bab, $abab$. Les facteurs de u sont ϵ, a, b, ab, ba, aba, bab, $abab$.

Proposition 2

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

- Si u et u' sont deux préfixes de w, alors u est préfixe de u' ou inversement.
- 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.

- Supposons que u et u' sont deux préfixes de w. Il existe donc deux mots t et t' tels que $ut = u't' = w$. Sans perte de généralité, on pourra supposer $|u| \leq |u'|$. Montrons que u est un préfixe de u', par récurrence sur $|u|$.

  Tout d'abord : si $|u| = 0$, alors $u = \epsilon$ et u est un préfixe de u', d'où le résultat.

  Autrement : u et u' sont tous deux de longueur au moins 1. Ainsi, il existe deux lettres a et b et deux mots x et y tels que $u = ax$ et $u' = by$. Alors, comme $ut = u't'$, on a : $axt = byt'$. Ces deux mots étant égaux, leurs deux premières lettres sont égales, d'où on tire : $a = b$. Il en vient que $xt = yt'$. Ainsi, x et y sont préfixes d'un même mot.

  Mais rappelons-nous que $u = ax$ et $u' = by$ donc x et y sont de longueur inférieure (strictement) à u et u'. On peut donc conclure par hypothèse de récurrence que x est préfixe de y. Donc ax est préfixe de $ay = by$, ie u est préfixe de u', d'où le résultat.
- Laissé en exercice. (Utiliser que si $uv = u'v'$, alors u et u' sont préfixes d'un même mot.)

□

Remarques :

- 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 relation d'ordre.

Définition 5

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 $u_i = v_{\varphi(i)} \quad \forall i \in [\![1; |u|]\!]$.

Remarques :

- u et ϵ sont toujours des sous-mots de u.
- ⚠Ne pas confondre sous-mot et facteur ! Un facteur est un sous-mot, mais un sous-mot n'est pas nécessairement un facteur.

Exemple : Les sous-mots de $abcb$ sont : ϵ, a, b, c, ab, ac, bc, bb, cb, abc, abb, acb, bcb, $abcb$.

II - Compressions et décompressions

II.1 - Principe général

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

Définition 6

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, $|comp(m)| < |m|$.

Remarques :

— 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

Définition 7

Compression avec perte (culture générale) :
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), ...

II.2 - Code binaire

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

Définition 8

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 9

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) En machine, les caractères ASCII sont représentés par des petits entiers, c'est-à-dire par des chaînes de 8 bits. L'encodage ASCII est donc un code binaire !

Sur combien de bits la chaîne de caractères ASCII "magicienne" est-elle représentée en machine ? → on utilise 8 bits pour chacun des 10 caractères, soit 80 bits au total.

Définition 10

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 ?

(Vous devriez réussir à encoder "magicienne" avec 30 bits : comme il y a 7 caractères différents, il faut au moins 3 bits par caractère pour pouvoir les représenter de manière distincte.)

Comment faire mieux ? Nous avons atteint la limite des codes de taille fixe. Il faut s'essayer au code de taille variable ! Quel problème peut-il alors se poser ?

Définition 11

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 ?
(Comparer l'encodage de abc et de bca...)

Définition 12

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 : Constater que e code de l'exemple précédent n'est pas préfixe.

Proposition 3

Tout code préfixe est uniquement déchiffrable.

Démonstration. Soit f un code préfixe sur Σ. Soient $u = u_1 \dots u_n \in \Sigma^*$ et $v = v_1 \dots v_p \in \Sigma^*$ tels que $f(u) = f(v)$. Montrons que nécessairement : $u = v$. On procède par récurrence sur $n = |u|$.

- Si $n = 0$: alors $u = \epsilon$ donc $f(u) = f(\epsilon) = \epsilon$, donc $f(v) = \epsilon$ et nécessairement $v = \epsilon$. On a donc bien : $u = v$.
- Sinon : par défintion d'un code binaire, on a $f(u) = f(u_1)f(u_2 \dots u_n)$ et $f(v) = f(v_1)f(v_2 \dots v_p)$. De plus $f(u) = f(v)$. Donc $f(u_1)$ et $f(v_1)$ sont préfixes d'un même mot, donc $f(u_1)$ est préfixe de $f(v_1)$ ou inversement. Comme f est un code préfixe, cela implique par définition que $u_1 = v_1$. Il reste donc que $f(u_2 \dots u_n) = f(v_2 \dots v_p)$, et comme ces mots sont de taille plus petite que u et v on conclut par récurrence que $u_2 \dots u_n = v_2 \dots v_p$.
  Finalement, on en tire $u = v$

Donc tout code préfixe est bien uniquement déchiffrable. □

Définition 13

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.

II.3 - Algorithme de Huffman

Idée générale :

Lorsqu'un caractère apparaît plus souvent que les autres dans le texte, on souhaite lui donner un encodage plus court. En contrepartie, ce sont les caractères les moins fréquents qui utiliseront plus de bits pour leur encodage.

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.

Définition 14

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).

Pour compresser un texte : on associe à chaque lettre c la chaîne de bits définie par le chemin allant de la racine à la feuille c dans l'arbre A.

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 :

Pour décompresser le code d'un texte compressé : on le code compressé jusqu'à reconnaître une chaîne de bits qui est l'encodage $f(c)$ d'un caractère c (en suivant dans l'arbre A le chemin étiqueté par le code que l'on lit jusqu'à atteindre une feuille).

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.

Il construit l'arbre du code via une stratégie gloutonne. On démarre avec pour chaque caractère c un arbre réduit à un unique noeud d'étiquette c. Un tel arbre est également associé au nombre d'occurrences du caractère c dans le texte à compresser.

À chaque étape :

- On choisit deux arbres associés aux nombres d'occurrences les plus faibles.
- On les fusionne en un seul arbre binaire (en construisant un nouveau noeud dont les deux enfants sont les arbres à fusionner).
- Ce nouvel arbre est associé à la somme des nombres d'occurrences des caractères des deux arbres fusionnés.

On procède ainsi de fusion en fusion jusqu'à ce qu'il ne reste qu'un unique arbre contenant tous les caractères.

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 :

Proposition 4 – Théorème 1

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.

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

Démonstration. Si un noeud dans l'arbre ne possède qu'un seul fils, on peut obtenir un nouvel arbre binaire en supprimant ce noeud et en faisant remonter son fils à sa place. Certains feuilles du nouvel arbre sont à une profondeur stricte strictement plus petite qu'avant, et sont ainsi associées à des chaînes de bits plus courtes, d'où un meilleur code. □

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] :

Soit f un code optimal associé à un arbre A_f. Soient a_i et a_j deux feuilles de profondeur maximale étant soeurs dans A_f (elles existent car A_f est strict). Montrons qu'en échangeant a_i et a_j avec les feuilles a_1 et a_2, le nouveau code obtenu reste optimal.

Appelons f' le code obtenu après échange de a_i avec a_1. Alors :

$$w_m(f') = w_m(f) + n_1(|f'(a_1)| - |f(a_1)|) + n_i(|f'(a_i)| - |f(a_i)|)$$

$$w_m(f') = w_m(f) + n_1(|f(a_i)| - |f'(a_i)|) + n_i(|f'(a_i)| - |f(a_i)|)$$

$$w_m(f') = w_m(f) + (n_1 - n_i)(|f(a_i)| - |f'(a_i)|)$$

Or, $(n_1 - n_i) \leq 0$ et $(|f(a_i)| - |f'(a_i)|) \geq 0$.

On a donc : $w_m(f') \leq w_m(f)$.

De même, en notant f'' le code obtenu à partir de f' en échangeant les feuilles a_j et a_2, on montre que $w_m(f'') \leq w_m(f')$.

Finalement : $w_m(f'') \leq w_m(f)$, et donc par optimalité de f, f'' est optimal également.

Il existe donc bien un code optimal dans lequel les feuilles a_1 et a_2 sont soeurs.

— **(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 : $\boxed{opt(\Sigma_m) \geqslant opt(\Sigma'_m) + n_1 + n_2}$.

Cette fois, en partant d'un code f optimal dans lequel a_1 et a_2 sont soeurs, notons f' le code obtenu en remplaçant les deux feuilles a_1, a_2 et leur parent par un seul noeud d'étiquette b.

On a :

$$w_m(f') = w_m(f) - n_1|f(a_1)| - n_2|f(a_2)| + (n_1 + n_2)|f'(b)|$$

$$w_m(f') = w_m(f) - (n_1 + n_2)|f(a_1)| + (n_1 + n_2)|f(a_1) - 1|$$

$$w_m(f') = w_m(f) - (n_1 + n_2) = opt(\Sigma_m) - (n_1 + n_2)$$

Or par définition : $w_m(f') \geq opt(\Sigma'_m)$.

Donc :

$$opt(\Sigma_m) \geq (n_1 + n_2) + opt(\Sigma'_m)$$

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

On procède par récurrence sur le cardinal p de l'alphabet. Si $p = 2$, c'est bon (on encode chaque caractère par un seul bit).

Si $p \geq 3$: notons $|\Sigma_m| = \{(a_1, n_1), \dots, (a_p, n_p)\}$ et $|\Sigma'_m| = \{(b, n_1 + n_2), (a_3, n_3), \dots, (a_p, n_p)\}$. L'algorithme de Huffman revient à :

— remplacer Σ_m par Σ'_m

— calculer le code de Huffman pour Σ'_m (on le note h')

— prendre l'arbre de h' et y remplacer la feuille d'étiquette b par un noeud ayant pour enfants les feuilles d'étiquette a_1 et a_2 (on note h ce nouveau code).

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

Alors : $w_m(h) = w_m(h') + n_1 + n_2$.

Or par hypothèse de récurrence, Huffman est optimal sur Σ'_m, donc $w_m(h') = opt(\Sigma'_m)$. Donc par lemme : $w_m(h) = opt(\Sigma'_m) + n_1 + n_2 \leq opt(\Sigma_m)$.

D'où $w_m(h) = opt(\Sigma_m)$, d'où l'optimalité de Huffman.

□

Remarques :

— Le codage de Huffman nécessite de transmettre l'arbre du code en plus du texte compressé (pour pouvoir décompresser ensuite). 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.

II.4 - Algorithme de Lempel-Ziv-Welch

Idée générale : On retient les motifs déjà rencontrés au fur et à mesure de la lecture du texte, afin de pouvoir compresser ces motifs plus efficacement s'ils réapparaissent par la suite.

Remarques : 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 $\rightarrow$ 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 $\{E, D, N, T\}$, compléter la compression le mot "ENTENDENT" par l'algorithme LZW, pas à pas.

étape	associations motif $\rightarrow$ code	texte à lire	résultat de la compression
début	E $\mapsto 0$; D $\mapsto 1$; $N \mapsto 2$; $T \mapsto 3$	ENTENDENT	0 (on compresse E)
	E $\mapsto 0$; D $\mapsto 1$; $N \mapsto 2$; $T \mapsto 3$; $EN \mapsto 4$	NTENDENT	02 (on compresse N)
	E $\mapsto 0$; D $\mapsto 1$; $N \mapsto 2$; $T \mapsto 3$; $EN \mapsto 4$; $NT \mapsto 5$		
	E $\mapsto 0$; D $\mapsto 1$; $N \mapsto 2$; $T \mapsto 3$; ...		
	E $\mapsto 0$; D $\mapsto 1$; $N \mapsto 2$; $T \mapsto 3$; ...		
	E $\mapsto 0$; D $\mapsto 1$; $N \mapsto 2$; $T \mapsto 3$; ...		
	E $\mapsto 0$; D $\mapsto 1$; $N \mapsto 2$; $T \mapsto 3$; ...		
	E $\mapsto 0$; D $\mapsto 1$; $N \mapsto 2$; $T \mapsto 3$; ...		

Pour retenir les associations motif $\to$ 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 $\to$ 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 $\to$ motif à chaque nouveau motif décompressé (avec un motif de retard par rapport à la compression).

Exemple : Décompresser le code obtenu précédemment pour "ENTENDENT", pas à pas :

étape	associations code $\to$ motif	code compressé	résultat de la décompression
début	$0 \mapsto$ E ; $1 \mapsto$ D ; $2 \mapsto$ N ; $3 \mapsto$ T	0 2 3 4 1 4 3	
	$0 \mapsto$ E ; $1 \mapsto$ D ; $2 \mapsto$ N ; $3 \mapsto$ T		
	$0 \mapsto$ E ; $1 \mapsto$ D ; $2 \mapsto$ N ; $3 \mapsto$ T		
	$0 \mapsto$ E ; $1 \mapsto$ D ; $2 \mapsto$ N ; $3 \mapsto$ T		
	$0 \mapsto$ E ; $1 \mapsto$ D ; $2 \mapsto$ N ; $3 \mapsto$ T		
	$0 \mapsto$ E ; $1 \mapsto$ D ; $2 \mapsto$ N ; $3 \mapsto$ T		
	$0 \mapsto$ E ; $1 \mapsto$ D ; $2 \mapsto$ N ; $3 \mapsto$ T		
	$0 \mapsto$ E ; $1 \mapsto$ D ; $2 \mapsto$ N ; $3 \mapsto$ 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.

— Décompresser le code obtenu :

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

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

III - Recherche dans un texte

Objectif : Chercher une chaîne de caractères donnée dans un texte (plus long).
Exemples :

— C'est ce que fait le Ctrl-F de votre éditeur ou navigateur préféré.

— Alignement de séquences en génétique.

Définition 15

- Soient $m = m_0 \dots m_{M-1}$ et $t = t_0 \dots t_{N-1}$ deux mots sur un même alphabet Σ. Une **occurrence** de m dans t est un entier i tel que $t_i \dots t_{i+M-1} = m$.
- Le problème de la **recherche d'occurrences** consiste, étant donnés m et t, a déterminer toutes les occurrences i de m dans t.

Remarque : Dans la suite, on note `t[i:j]` le facteur de `t` compris entre les indices i inclus et j exclus : `t[i:j]` = $t_i \dots t_{j-1}$. Vous DEVEZ réintroduire cette notation si vous souhaitez l'utiliser dans une copie !
Remarques :

— La chaîne de caractères vide "" de longueur 0 apparait à toutes les positions du mot. On a donc $N+1$ occurrences du mot vide dans un texte t de longueur N.

— ⚠ Deux occurrences peuvent se chevaucher. Exemple : le mot *ana* apparait deux fois dans *ananas*.

Soulignez ces deux occurrences dans le mot : a n a n a s

— On peut étendre le problème de la recherche d'occurrences à des variantes plus larges. Par exemple, on peut vouloir chercher un motif général plutôt qu'une chaîne précise, comme les motifs de la forme `*.*@*.*` où `*` représente n'importe quelle suite de lettres. Nous verrons ceci beaucoup plus en détail en MPI avec les expressions régulières.
On peut encore, par exemple, chercher simultanément les occurrences d'un ensemble fini de mots dans un texte, plus efficacement qu'avec une recherche distincte pour chaque mot.

III.1 - Algorithme naïf

Exercice : Proposer un algorithme naïf pour résoudre le problème de la recherche d'occurrences (simple description en Français) :

Pour chaque position possible dans le mot t, on regarde si m est un facteur de t à cette position. Autrement dit, pour tout i entre 0 et N-M, on teste si t[i:i+M] est égal à m.

C : recherche-naif.c

```c
int naive_string_search(char *m, char *t) {
    int M = strlen(m), N = strlen(t), count = 0;
    for (int i = 0; i <= N - M; i += 1)
        if (strncmp(m, &t[i], M) == 0) { //strncmp compare les M premiers caractères
        //des deux chaînes données en argument
            printf("occurrence à la position %d\n", i);
            count+=1;
        }
    return count;
}
```

Explication [2] :

La fonction `strlen` permet de récupérer la longueur d'une chaîne de caractères.

`strncmp` compare deux chaînes de caractères, pour l'ordre lexicographique, dans la limite d'un nombre de caractères donné (ici M). Elle renvoie -1 ou 1 si elle observe une différence entre les deux chaînes, à savoir -1 si la première est plus petite et 1 si elle est plus grande, et 0 si les deux chaînes contiennent au moins M caractères chacune, deux à deux égaux.

Par exemple, `strncmp("foo", "fool", 3)` renvoie 0, là où `strcmp("foo", "fool")` renverrait -1. Si la fonction `strncmp` observe une première différence à l'indice j, elle aura comparé $j+1$ caractères. Si elle renvoie 0, elle aura comparé M caractères.

2. citation de *Informatique - MP2I/MPI* de Filliâtre et. al

Complexités : La comparaison de deux chaînes de caractères de longueur M (via `strncmp`) se fait en temps $O(M)$. Tant que les caractères sont égaux, on continue à lire les deux mots jusqu'à trouver un caractère différent ou conclure que les deux mots sont égaux.

On effectue $N - M + 1$ itérations (avec $M \ll N$), et chaque itération en temps $O(M)$. Au total, on a donc un temps $O(NM)$ pour l'algorithme naïf.

Le pire cas est atteint lorsque, par exemple, on cherche le motif *a...a* constitué de M fois la lettre 'a' (ou encore le motif *a...ab*) dans le texte *a...a...a* consituté de $N \gg M$ fois la lettre 'a'.

La complexité spatiale est constante, on définit un nombre borné de variables dans la fonction et on se contente de lire les caractères données en entrée.

III.2 - Algorithme de Rabin-Karp

Idée générale :

Limiter les comparaisons de mots (coûteuses) en commençant par comparer un autre critère plus rapide à calculer sur les deux chaînes. On compare une empreinte (un hash) du facteur t[i:i+M] avec l'empreinte du mot m cherché, en temps O(1). - Si les empreintes diffèrent, on est sûr qu'il n'y a pas d'occurrence en i (les mots ne peuvent pas être égaux) et on passe à la position suivante. - Si elles coïncident, on compare les chaînes entières pour vérifier qu'il ne s'agit pas d'une collision entre deux éléments différents.

Dans l'idée, on a envie de faire ça (mais ce code est aussi mauvais que le naïf) :

Pseudo-code

```
//Algorithme FAUX donnant l'idée générale de Rabin-Karp
Entrée : mots m et t
Sortie : occurrences de m dans t
RK_FAUX(m, t) :
    N ← |t|
    M ← |m|
    cible ← hash(m)
    occurences ← {}
    Pour i allant de 0 à N − M Faire :
        Si hash(t[i : i + M]) = cible && t[i : i + M] = m Alors :
            occurrences ← occurrences ∪ {i}
    Renvoyer occurrences
```

Contraintes :

Pour que l'algorithme soit efficace, il faut que :

- le calcul de l'empreinte soit rapide ($O(1)$)
- il y ait peu de collisions (pourcentage faible) des entrées

Ces deux objectifs sont en conflit : pour limiter les collisions, il est indispensable que l'empreinte dépende de chacun des M caractères du motif considéré. Sinon, on pourrait modifier librement les caractères qui ne sont pas pris en compte par la fonction de hachage, et on obtiendrait autant de collisions que de possibilités.

Mais le calcul de l'empreinte doit être en $O(1)$, ce qui est impossible si le calcul doit dépendre de M caractères indépendants. Il faut au moins les lire en temps $O(M)$. On obtiendrait donc au moins une complexité proportionnelle à M, donc du $O(NM)$ en temps au total, pas mieux que le naïf...

Comment régler ce conflit ?

Remarquons que les calculs successifs d'empreintes ne sont pas indépendants ! Les facteurs successifs dont un calcule les empreintes ont toujours $M - 1$ caractères en commun. On va exploiter ces ressemblances.

Définition 16

On utilise une **empreinte tournante** (*rolling hash* en Anglais). On choisit une fonction de hachage qui permet de calculer en temps constant l'empreinte de $t_i \dots t_j$ en connaissant l'empreinte de $t_{i-1} \dots t_{j-1}$.

Proposition 5

Soit $b \geqslant 2$ une base entière. La fonction de hachage h suivante convient :
$h(x_0 \dots x_{k-1}) = x_0 b^{k-1} + x_1 b^{k-2} + \dots x_{k-1} b^0$.
i.e. $h(x_0 \dots x_{k-1})$ est l'écriture de $x_0 \dots x_{k-1}$ dans la base b, avec le chiffre le plus significatif (de poids fort) en premier.

Remarques :

— Il faut que le poids fort (base b) soit sur le chiffre qu'on va "enlever" au prochain calcul d'empreinte, donc nécessairement sur le premier chiffre.

— On voit nos caractères comme des entiers (souvent compris entre 0 et 255 : les caractères ASCII). La base b doit vérifier $b \geqslant 256$ (plus grande que le plus grand des x_i) pour éviter entièrement les collisions.

Cette fonction de hachage h prise telle quelle pose un problème : elle renvoie des valeurs non bornées, potentiellement très élevées quand M est grand. Pour $b = 2^8 = 256$, on dépasse la capacité d'un entier 64 bits dès que $M = 7$...

On va donc travailler modulo un certain entier p, avec quelques collisions. On choisit p premier et p grand pour limiter les collisions, mais pas trop grand pour éviter les dépassements de capacité dans les calculs intermédiaires.

Algorithme de Rabin-Karp :

Pseudo-code

```
//On fixe a priori une base b et un nombre premier p,
//légèrement inférieur au plus grand entier machine.
Entrée : mots m et t
Sortie : occurrences de m dans t
RABIN-KARP(m, t) :
    N ← |t|
    M ← |m|
    Si M > N Alors :
        Renvoyer {}
    d ← b^(k-1) mod p //calcul intermédiare stocké (utile au calcul des nouveaux hash)
    cible ← 0 //empreinte de m
    h ← 0 //empreinte tournante de t[i:i+M]
    occurences ← {}
    //calcul initial des empreintes (hash(m) et hash(t[0:M])):
    Pour i allant de 0 à M-1 Faire :
        cible ← b.cible + m[i] mod p
        h ← b.h + t[i] mod p
    //parcours du texte :
    Pour i allant de 0 à N-M Faire :
        Si h = cible et t[i : i + M] = m Alors :
            occurrences ← occurrences ∪ {i}
        Si i + M < N Alors :
            h ← b.(h - d.t[i]) + t[i + M] mod p
    Renvoyer occurrences
```

Remarques :

— Pour être sûr d'éviter les dépassements de capacité, il faut que $p.b$ soit représentable dans un entier.

— En moyenne, sur des données aléatoires, on aura n/p collisions environ.

Remarque :

— Rabin-Karp n'est pas l'algorithme le plus efficace pour rechercher une chaîne dans un texte, mais il est simple à implémenter et s'étend facilement à la recherche d'un ensemble S de motifs dans un texte. On calcule initialement l'empreinte des différents motifs de S, et on remplace le test d'égalité des empreintes par le test de $h \in S$.

Un exemple "à la main"

Faire tourner l'algorithme de Rabin-Karp sur une exemple, pas à pas. On considère le texte *affiner* dans lequel on recherche le motif *fin*.

Pour simplifier les calculs à nos capacités humaines, on prend la fonction de hachage tournante suivante :

Pour calculer $h(x_1 \dots x_n)$, on convertit chacune des lettres x_i en entier entre 1 et 26 correspondant à sa position dans l'alphabet, puis on prend le tout modulo 10. L'empreinte de *fin* est donc $6 + 9 + 14 = 29$ pris modulo 10, c'est à dire 9.

étape i	`cible` (hash de ”fin”)	`h` (hash de t[i:i+3])	comparaison de chaînes	Explication :
0 : aff iner	9			
1 : a ffi ner	9			
2 : af fin er	9			
3 : aff ine r	9			
4 : affi ner	9			

III.3 - Algorithme de Boyer-Moore

Présentons un autre algorithme pour résoudre le problème de la recherhe d'un mot m dans un texte t : l'algorithme de Boyer-Moore. Dans ce cours, nous ne présenterons qu'une version simplifiée de cet algorithme (avec une seule table de décalage), appelée algorithme de Boyer-Moore-Horspool.

Rappel : l'algorithme naïf compare les deux chaînes de caractères m et $t[i : i + M]$ pour chaque position $0 \leqslant i \leqslant N - M$ du texte.

Idée générale :
On continue de tester la présence de m à la position i dans t, de gauche à droite. Mais au lieu de tester tous les alignements possibles, on va sauter les tests de certaines positions i qu'on sait déjà impossible grâce aux comparaisons qu'on vient d'effectuer.
Détaillons :

- À chaque position i qu'on teste, on compare les deux chaînes m et $t[i : i+M]$ caractère par caractère de droite à gauche (en partant du dernier caractère). C'est à dire qu'on compare d'abord $m[M-1]$ et $t[i+M-1]$.
- Si tous les caractères coïncident, on a trouvé une occurrence. Sinon, on arrête les comparaisons au premier caractère différent trouvé. Notons j la position de ce caractère, c'est-à-dire le plus grand entier tel que $0 \leqslant j < M$ et $m[j] \neq t[i+j]$. Notons c le caractère différent $t[i+j]$.

Lorsqu'on cherchera une occurrence du mot m plus loin (par exemple à la position i+1), cela reviendra à décaler le mot m vers la droite sous le mot t (par exemple de 1 case). Or on sait que m n'est une occurrence dans t à ces positions que si les caractères de m et t coïncident, donc sous le caractère c à la position $i + j$, il faudra qu'il y a une lettre c apparaissant dans m. L'algorithme de Boyer-Moore-Horspool consiste donc à avancer le mot vers la droite d'autant de positions qu'il faut pour aligner un c dans m avec le c lu dans t.

- S'il y a plusieurs fois la lettre c dans m avant l'indice j, on s'arrête à celle qui décale le mot le moins possible (pour être sûr de ne rater aucune occurrence possible du mot dans le texte). Autrement dit, on effectue l'affectation :

$$i \leftarrow i + j - k_c \quad \text{avec} \quad k_c = max\{k \in [\![0; j]\!] \ \mid \ m[k] = c\}$$

- Si la lettre c n'apparaît pas du tout dans le mot m avant l'indice j, alors on peut avancer le mot vers la droite jusqu'à le faire commencer à droite du caractère c, car on sait que c'est impossible l'aligner avec c. Autrement dit, on effectue l'affectation :

$$i \leftarrow i + j + 1$$

Exemple :
Faire tourner l'algorithme de Boyer-Moore-Horspool à la main, pas à pas, sur l'exemple suivant :
On recherche le motif ”abaaa” dans le texte ”abcaababbaabaaaa”.

i = 0 :

a	b	c	a	a	b	a	b	b	a	a	b	a	a	a	a
a	b	a	a	a											

i = :

a	b	c	a	a	b	a	b	b	a	a	b	a	a	a	a

i =   **:**

a	b	c	a	a	b	a	b	b	a	a	b	a	a	a	a

i =   **:**

a	b	c	a	a	b	a	b	b	a	a	b	a	a	a	a

i =   **:**

a	b	c	a	a	b	a	b	b	a	a	b	a	a	a	a

i =   **:**

a	b	c	a	a	b	a	b	b	a	a	b	a	a	a	a

i =   **:**

a	b	c	a	a	b	a	b	b	a	a	b	a	a	a	a

Mise en oeuvre : Plutôt que de rechercher l'indice k_c à chaque étape (en temps $O(j)$), on peut précalculer une **table de décalages** qui nous donne directement (en accès $O(1)$) l'entier k_c recherché pour tout position j dans m et tout caractère c possibles. Les décalages ne dépendent pas du texte lui-même.

Autrement dit, la case indexée par l'entier j, $0 \leqslant j < M$ et par le caractère $c \in \Sigma$ contient le plus grand entier k tel que $0 \leqslant k < j$ et $m[k] = c$ s'il existe, et rien sinon.

Exemple :
Donnons la table de décalage pour le motif ”abracadabra” (ne dépend que du mot, pas du texte) :

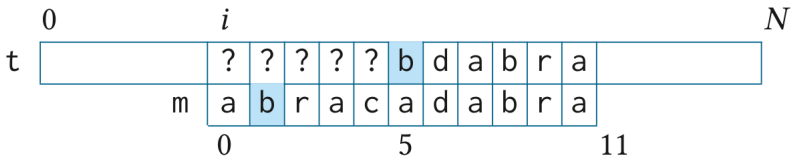
	a	b	r	c	d
0					
1	0				
2	0	1			
3	0	1	2		
4	3	1	2		
5	3	1	2	4	
6	5	1	2	4	
7	5	1	2	4	6
8	7	1	2	4	6
9	7	8	2	4	6
10	7	8	9	4	6

Table de décalages pour le motif ”abracadabra” (M = 11)

Prenez le temps de lire et de comprendre cette table de décalage.

Exercice :

1. En utilisant la table de décalage, trouver de combien il faut avancer l'indice i dans la situation suivante :



C : boyer-moore.c

```c
//Entrée : deux mots m et t (comme d'habitude...)
//Sortie : nombre d'occurrences de m dans t
int boyer_moore(char *m, char *t) {
    int M = strlen(m), N = strlen(t);
    int **table = build_table(N, M); /* fonction intermédiaire qui
                                        construit la table des décalage (laissée en exercice) */
    int count = 0;
    for (int i = 0; i <= N - M;) { //notez qu'on ne décale pas i de 1 entre les itérations

        //comparaison des chaînes m et t[i:i+M] de droite à gauche :
        int decalage = 0;
        for (int j = M - 1; j >= 0; j-=1) {
            if (t[i + j] != m[j]) {
                unsigned char c = t[i + j];
                decalage = j - table[j][c];
                break;
            }
        }
        //lettre absente du mot m :
        if (decalage == 0) {
            printf("occurrence à la position %d\n", i);
            count++;
            decalage = 1;
        }
        i += decalage; //RARE exemple de modification volontaire de i
        //dans une boucle Pour sur i
    }
    return count;
}
```