Algorithmique du texte

Guillaume Rousseau
MP2I Lycée Pierre de Fermat
guillaume.rousseau@ens-lyon.fr

16 juin 2024

L'algorithmique du texte permet d'étudier des problèmes ayant de nombreuses applications, dans des logiciels de la vie de tous les jours comme dans des domaines poussés de la recherche.

Trois exemples de problèmes à l'importance fondamentale sont :

— Mesurer la similarité entre deux textes

— Rechercher un motif dans un texte

— Comprimer un texte afin de réduire sa taille en mémoire

Nous avons déjà vu un exemple d'algorithme qui permet de mesurer la similarité entre deux textes lorsque nous avons étudié la programmation dynamique : la distance de Levenshtein.

Rappel : distance de Levenshtein Soit Σ un alphabet, et $u, v \in \Sigma^*$ deux mots. La distance de Levenshtein entre u et v est le plus petit nombre d'opérations à appliquer pour transformer u en v parmis les trois suivantes :

— Insérer une lettre dans un mot

— Supprimer une lettre d'un mot

— Transformer une lettre en une autre

On peut également assigner un coût à chaque opération, et même assigner un coût à chaque opération selon la lettre insérée, supprimée ou transformée. Ceci permet de modéliser la similarité entre deux textes en prenant en compte des aspects phonétiques, orthographiques, ou même pratiques (deux lettres proches sur le clavier AZERTY peuvent être facilement interverties dans un mot).

Dans ce chapitre, on se focalise sur les deux derniers problèmes évoqués plus haut : la recherche de motif et la compression de données.

1 Recherche de motif

Le problème le plus simple de recherche de motif est de savoir, étant donné un texte t et un plus petit texte m (appelé motif), si m apparaît tel quel dans t.

Problème 1 : MOTIF

Entrée(s) : $t = t_0 t_1 \dots t_{n-1}, m = m_0 \dots m_{p-1} \in \Sigma^*$

Sortie(s) : Existe t-il un indice $i \in [\![0, n-1]\!]$ tel que $t_i t_{i+1} \dots t_{t_i+p-1} = m$

A Première approche : force brute

L'algorithme le plus naturel pour résoudre de problème est de simplement tester toutes les possibilités pour l'indice de début du motif :

Algorithme 2 : Recherche de motif : méthode naïve

Entrée(s) : $t, m \in \Sigma^*$ avec $|t| = n$, $|m| = p$

Sortie(s) : i tel que m apparaît à la position i de t

```
1 pour i = 0 à n − p faire
2     j ← 0 tant que j < p et t_{i+j} = m_j faire
3         j ← j + 1;
4     si j = p alors
5         retourner i
6 retourner Pas d'occurence
```

La complexité est en $\mathcal{O}(np)$.

Définition 1. Lors de la recherche d'un motif m de taille p dans un texte $t = t_0 \dots t_{n-1}$ de taille n, pour $i \in [\![0, n-p]\!]$, la **fenêtre de t à la position** i est le mot $t_i \dots t_{i+p-1}$. C'est donc la partie du texte que l'on compare avec le motif à partir de la position i.

Exercice 1. Donner un exemple de texte et de motif atteignant le pire cas, $\Theta(np)$ opérations. Donner un autre exemple pour lequel le coût n'est que de $\mathcal{O}(n+p)$.

Certaines variantes du problème demandent de trouver la liste de toutes les occurrences, ce que l'on peut faire facilement en adaptant l'algorithme naïf précédent.
Dans le reste de cette section, on étudie deux algorithmes permettant d'améliorer cette méthode.

1. L'algorithme de Boyer-Moore consiste à pré-traiter le motif pour construire des tables permettant de faire sauter i vers l'avant dans certains cas, et donc d'accélérer la recherche.
2. L'algorithme de Rabin-Karp consiste à utiliser une bonne fonction de hachage, permettant de ne comparer que les hash des mots plutôt que les mots en entier.

B Algorithme de Boyer-Moore

Commençons par modifier l'algorithme naïf pour qu'il compare le motif et les portions du texte en commençant à chaque fois par la dernière lettre du motif :

Algorithme 3 : Recherche de motif : méthode naïve 2

Entrée(s) : $t, m \in \Sigma^*$ avec $|t| = n$, $|m| = p$
Sortie(s) : i tel que m apparaît à la position i de t

```
1 pour i = 0 à n − p faire
2     j ← p − 1 tant que j ≥ 0 et t_{i+j} = m_j faire
3         j ← j − 1;
4     si j = −1 alors
5         retourner i
6 retourner Pas d'occurrence
```

Appliquons cet algorithme sur un exemple : $t =$ `BABACACABAAAAC` et $m =$ `BAAAA`. Pour $i = 0$, on compare le motif `BAAAA` et la portion du texte `BABAC`. On compare les dernières lettres des deux mots : A et C.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
B	A	B	A	C	A	C	A	B	A	A	A	A	C
B	A	A	A	A									

On sait donc que le motif ne se trouve pas à la position $i = 0$. Mais on peut extraire une autre information : le motif ne contient pas C, et donc le motif ne peut pas commencer à $i = 0, 1, 2, 3, 4$.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
B	A	B	A	C	A	C	A	B	A	A	A	A	C
B	A	A	A	A									
	B	A	A	A	A								
		B	A	A	A	A							
			B	A	A	A	A						
				B	A	A	A	A					

On peut donc directement augmenter i à 5 et recommencer. On compare donc le motif `BAAAA` et la portion du texte `ACABA`. On compare les dernières lettres des deux mots : A et A. Les deux lettres concordent, on passe aux lettres suivantes : A et B.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
B	A	B	A	C	A	C	A	B	A	A	A	A	C
					B	A	A	A	A				

Les deux lettres sont différentes, donc on sait que le motif ne se trouve pas à la position $i = 5$. On ne peut pas appliquer directement le raisonnement précédent, car le motif contient bien la lettre B. Cependant, on voit que le seul endroit où le motif a un B est la lettre m_0. Donc, le motif ne peut pas se trouver aux positions $i = 5, 6, 7$:

0	1	2	3	4	5	6	7	8	9	10	11	12	13
B	A	B	A	C	A	C	A	B	A	A	A	A	C
					B	A	A	A	A				
						B	A	A	A	A			
							B	A	A	A	A		

On peut donc tester directement $i = 8$, et en comparant lettre par lettre, on trouve bien le motif à cet emplacement :

0	1	2	3	4	5	6	7	8	9	10	11	12	13
B	A	B	A	C	A	C	A	B	A	A	A	A	C
								B	A	A	A	A	

Ces remarques forment la première partie de l'algorithme de Boyer-Moore : la **règle du mauvais caractère**.

Définition 2. Soit $m \in \Sigma^*$ un motif de taille p et $a \in \Sigma$. La dernière occurence non-finale de a dans m, notée $d_m(a)$, est définie par :

$$d_m(a) = \max\{i \in [\![0, p-2]\!] \mid m_i = a\}$$

en prenant comme convention $\max \emptyset = -1$.
C'est donc le dernier indice où a apparaît dans m, ou bien l'avant dernier si m se termine par a.

Proposition 1 (Règle du mauvais caractère)**.** Soit $t = t_0 \dots t_{n-1}$ un texte et $m = m_0 \dots m_{p-1}$ un motif. Soit $i \in [\![0, n-1]\!]$ et $j \in [\![0, p-1]\!]$ tels que $m_j \neq t_{i+j}$. Alors aucune fenêtre de t de position $i' \in [\![i, i+j-d_m(t_{i+1})-1]\!]$ n'est égale à m.

En reprenant les mêmes notations, la règle du mauvais caractère permet donc de faire directement l'indice de recherche de i à $i + j - d_m(t_{i+1})$.

Démonstration. On remarque que pour une lettre a, a n'apparaît pas dans $m_{d_m(a)} \dots m_{p-1}$, sauf éventuellement en m_{p-1}. Donc, lorsque l'on compare le motif m à la fenêtre $t_i, \dots t_{i+p-1}$, et que $x_j \neq t_{i+j}$, on sait qu'on ne pourra pas trouver le motif si l'on met t_{i+j} en face d'une position de m entre $d_m(t_{i+j})$ et j :

t_i	$\dots$	$t_{i+d_m(t_{i+j})-1}$	$t_{i+d_m(t_{i+j})}$	$t_{i+d_m(t_{i+j})+1}$	$\dots$	t_{i+j-1}	t_{i+j}	t_{i+j+1}	$\dots$	t_{i+p-1}
x_0	$\dots$	$x_{d_m(t_{i+j})-1}$	$x_{d_m(t_{i+j})}$	$x_{d_m(t_{i+j})+1}$	$\dots$	x_{j-1}	x_j	x_{j+1}	$\dots$	x_{p-1}

On peut donc directement tester de mettre t_{i+j} en face de $x_{d_m(i+j)}$. Cela revient à avancer le motif de $j - d_m(t_{i+j})$ places, i.e. de tester $i' = i + j - d_m(t_{i+j})$:

$t_{i+j-d_m(t_{i+j})}$	$\dots$	t_{i+j}	$\dots$
x_0	$\dots$	$x_{d_m(t_{i+j})}$	$\dots$

□

L'utilisation de cette règle seule donne l'algorithme de Boyer-Moore-Horspool :

Algorithme 4 : Recherche de motif : Boyer-Moore-Horspool

Entrée(s) : $t, m \in \Sigma^*$ avec $|t| = n$, $|m| = p$
Sortie(s) : i tel que m apparaît à la position i de t

```
1  i ← 0;
2  tant que i < n − p + 1 faire
3      j ← p − 1;
4      tant que j ≥ 0 et t_{i+j} = m_j faire
5          j ← j − 1;
6      si j = −1 alors
7          retourner i
8      sinon
9          i ← i + max(1, j − d_m(t_{i+j}));
10 retourner Pas d'occurence
```

Prétraitement Le calcul de $d_m(a)$ est coûteux : il est a priori en $O(p)$. Cependant, on peut pré-traiter m et calculer toutes les valeurs d'un coup puis les stocker.

Exercice 2. Proposer une structure de donnée pour calculer et stocker les valeurs de $d_m(a)$. Quelle est le coût de construction de cette structure ?

Exercice 3. Appliquer l'algorithme précédent sur l'exemple suivant :

$$t = \texttt{BADACBIAAABBIBIDIADCCBABIDIAA}$$
$$m = \texttt{BIBIDIBABIDI}$$

On commencera par calculer la table des $d_m(a)$ pour $a \in \Sigma$. On comptera le nombre exact de comparaisons faites entre deux lettres, et on comparera avec l'algorithme naïf.

L'algorithme de Boyer-Moore complet nécessite l'utilisation d'une deuxième règle : la règle du bon suffixe.

Regardons un exemple pour comprendre d'où vient cette règle et comment l'utiliser. On cherche le motif $m = \texttt{CBACABACBA}$ dans le texte $t = \texttt{CABCCABABACBACABACBAAC}$:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
C	A	B	C	C	A	B	A	B	A	C	B	A	C	A	B	A	C	B	A	A	C
C	B	A	C	A	B	A	C	B	A												

Les deux dernières lettres, `BA`, coincident, mais pas la troisème : C dans le motif et A dans le texte. On peut donc éliminer la position $i = 0$. De plus, on sait que le terme `BA` doit apparaître dans le motif à un autre endroit. On peut donc directement chercher la position la plus à droite de m où `BA` apparaît (excepté en tant que suffixe car c'est ce qu'on est en train de tester !), et aligner cette position avec le `BA` lu dans le texte. En effet, toutes les positions intermédiaires ne peuvent pas convenir car la partie `BA` du texte ne sera pas en face d'un `BA` dans le motif :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
C	A	B	C	C	A	B	A	B	A	C	B	A	C	A	B	A	C	B	A	A	C
C	B	A	C	A	B	A	C	B	A												
	C	B	A	C	A	B	A	C	B	A											
		C	B	A	C	A	B	A	C	B	A										
			C	B	A	C	A	B	A	C	B	A									

On peut donc tester directement la position $i = 3$: les 6 dernières lettres coincident, mais pas la 7ème. On a donc le terme `ABACBA` qui correspond. Il n'a pas d'autre occurence dans m. Donc, on peut directement décaler le début de m après le début de cette partie du texte, car aucune position intermédiaire ne permettra de matcher `ABACBA` :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
C	A	B	C	C	A	B	A	B	A	C	B	A	C	A	B	A	C	B	A	A	C
			C	B	A	C	A	B	A	C	B	A									
								C	B	A	C	A	B	A	C	B	A				

On peut même aller plus loin : si une position valide du motif se trouve au milieu du terme `ABACBA`, alors on a un préfixe de m qui est aussi un suffixe de `ABACBA`. On recherche alors le plus grand préfixe de m qui est aussi suffixe de `ABACBA`, et on trouve `CBA` :

A	B	A	C	B	A							
			C	B	A	C	A	B	A	C	B	A

On peut donc directement décaler le motif jusqu'à avoir mis en face les deux CBA du motif et du texte :

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
C	A	B	C	C	A	B	A	B	A	C	B	A	C	A	B	A	C	B	A	A	C
								C	B	A	C	A	B	A	C	B	A				
										C	B	A	C	A	B	A	C	B	A		

On teste donc directement la position $i = 10$, et on trouve une occurence du motif dans t.

Introduisons maintenant les notations qui vont permettre de décrire précisément cette règle du bon suffixe.

Définition 3. Soit $m \in \Sigma^*$ un motif de taille p. Pour $j \in [\![0, p-1]\!]$, on note :

$$s_m(j) = \max\{k \in [\![0, j-1]\!] \mid m_k \dots m_{k+p-j-1} = m_j \dots m_{p-1} \text{ et } m_{k-1} \neq m_{j-1}\}$$

Autrement dit, $s_m(j)$ est l'indice maximal où le terme $m_j \dots m_{p-1}$ apparaît dans m sans être précédé de m_{j-1}. On pose également $s_m(p) = p - 1$ par convention.

On note également :

$$p_m(j) = \max\{k \in [\![0, p-j]\!] \mid m_0 \dots m_{k-1} = m_{p-k} \dots m_{p-1}\}$$

Autrement dit, $p_m(j)$ est la longueur du plus long préfixe de m qui est suffixe de $m_j \dots m_{p-1}$. On pose également $p_m(0) = p_m(1)$ pour empêcher de considérer m tout entier.

Exercice 4. Donner les valeurs de s_m et p_m pour le mot $m =$ `CBACABACBA` :

j	0	1	2	3	4	5	6	7	8	9
m_j	C	B	A	C	A	B	A	C	B	A
$s_m(j)$										
$p_m(j)$										

On peut alors utiliser les valeurs de s_m et p_m pour accélérer la progression de i, comme dans l'exemple précédent, selon la règle suivante, appelée **règle du bon suffixe**.

Proposition 2 (Règle du bon suffixe)**.** Soit $t, m \in \Sigma^*$, soit $t_i \dots t_{i+p-1}$ une fenêtre de t que l'on compare à m. On suppose qu'il existe j tel que $m_j \neq t_{i+j}$ et tel que $m_k = t_{i+k}$ pour $k > j$. Alors :

(i) Si $s_m(j+1) \geq 0$, alors pour $i' \in [\![i, i + j - s_m(j+1)]\!]$, $m \neq t_{i'} \dots t_{i'+p-1}$

(ii) Sinon, alors pour $i' \in [\![i, i + p - p_m(j+1) - 1]\!]$, $m \neq t_{i'} \dots t_{i'+p-1}$

Cette propriété nous donne donc deux critères permettant de faire avancer i plus vite que l'algorithme naïf.

Démonstration. (i) Supposons $s_m(j+1) \geq 0$, et par l'absurde prenons $i' \in [\![i, i+j-s_m(j+1)]\!]$ tel que $m = t_{i'} \dots t_{i'+p-1}$. En particulier, on a $m_{i+j-i'+1} \dots m_{i+p-1-i'} = t_{i+j+1} \dots t_{i+p-1}$. Or, $t_{i+j+1} \dots t_{i+p-1} = m_{j+1} \dots m_{p-1}$: on a donc trouvé une occurence du suffixe $m_{j+1} \dots m_{p-1}$ de m à partir de $m_{i+j-i'+1}$. Autrement dit, $s_m(j+1) \geq i+j-i'+1$. Donc, $i' > i+j-s_m(j+1)$: c'est absurde.

(ii) Supposons $s_m(j+1) = -1$. Alors, le suffixe $m_{j+1} \dots m_{p-1}$ n'apparaît jamais dans m sans être précédé de m_j. Donc, comme pour le premier point, on sait que pour $i' \leq i+j$, $m \neq t_{i'} \dots t_{i'+p-1}$.

Montrons par l'absurde que pour $i' \in [\![i+j+1, i+p-p_m(j+1)-1]\!]$, $m \neq t_{i'} \dots t_{i'+p-1}$. Supposons donc que $m = t_{i'} \dots t_{i'+p-1}$. Alors en particulier, $t_{i'} \dots t_{i+p-1} = m_0 \dots m_{i+p-i'-1}$. Mais, $t_{i'} \dots t_{i+p-1}$ correspondait aussi à m lorsque l'on a comparé à partir de la position i. Autrement dit, $t_{i'} \dots t_{i+p-1} = m_{i'-i} \dots m_{p-1}$. On a donc un préfixe de m qui est aussi suffixe de $m_{i'-i} \dots m_{p-1}$, et donc a fortiori de $m_{j+1} \dots m_{p-1}$ car $j \leq i'-i$.

Donc, $p_m(j+1) \geq i+p-i'$, ce qui est absurde car $i' \leq i+p-p_m(j+1)-1$.

□

Nous pouvons maintenant décrire l'algorithme de Boyer-Moore, qui consiste à appliquer les deux règles précédentes (mauvais caractère et bon suffixe), et à choisir à chaque étape celle qui nous permet de sauter le plus de positions. Nous allons également modifier l'algorithme pour qu'il renvoie la liste de **toutes** les occurences du motif.

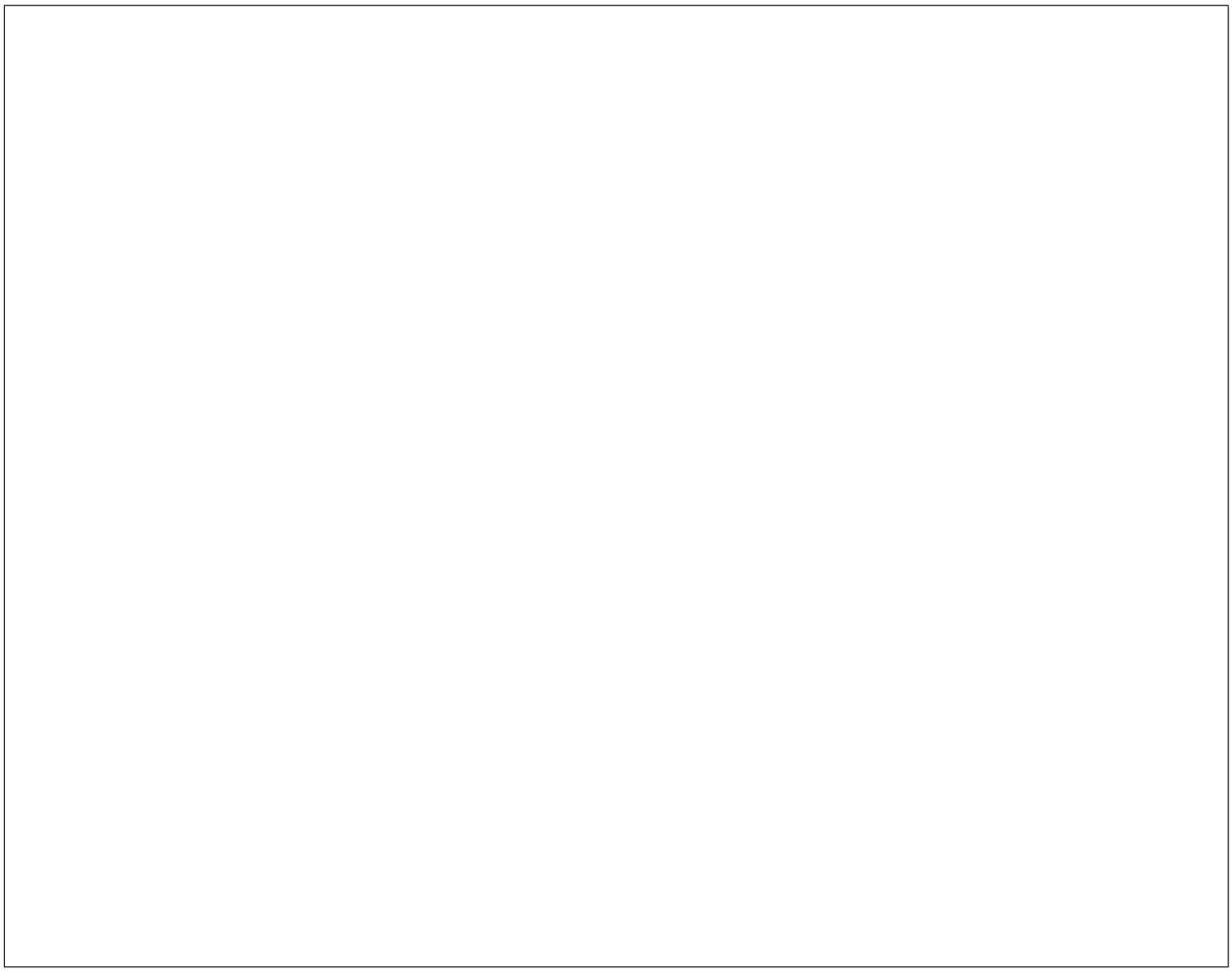
Algorithme 5 : Recherche de motif : Boyer-Moore

Entrée(s) : $t, m \in \Sigma^*$ avec $|t| = n$, $|m| = p$
Sortie(s) : Liste des indices i tel que m apparaît à la position i de t

```
1  D, P, S ← tableaux stockant d_m, p_m et s_m;
2  L ← [] // résultat de l'algorithme, liste des positions
3  i ← 0;
4  tant que i ≤ n − p faire
5      j ← p − 1;
6      tant que j ≥ 0 et t_{i+j} = m_j faire
7          j ← j − 1;
8      si j = −1 alors
9          Ajouter i à L;
10         i ← i + p − P[1];
11     sinon
12         si S[j + 1] ≥ 0 alors
13             i ← i + max(1, j − D[t_{i+j}], j + 1 − S[j + 1]);
14         sinon
15             i ← i + max(1, j − D[t_{i+j}], p − P[j + 1]);
16     retourner L
17 retourner Pas d'occurence
```

Exemple 1. Appliquer l'algorithme pour :

$t =$ `AABCCBABABCADABABADDABBABABCADABABABCADABABCBAD`
$m =$ `BABABCADABAB`



Complexité Le calcul de D se fait en $\mathcal{O}(p)$ et celui de P et S se fait assez facilement en $\mathcal{O}(p^2)$ (et peut même se faire en temps linéaire). Malgré l'utilisation de ces tables, dans le pire cas la recherche de toutes les occurrences prend un temps $\mathcal{O}(np)$. Donc, ce n'est pas mieux asymptotiquement que l'algorithme naïf ! Cependant, on peut montrer que pour trouver la première occurrence, la partie recherche s'exécute en $\mathcal{O}(n + p)$. De plus, en pratique, cet algorithme est très efficace, car les pire cas sont très rares. En fait, dans de nombreux cas d'usage, la règle du mauvais caractère seule suffit.

Remarquons que les tables D, P et S ne dépendent que du motif m. Ainsi, si l'on veut chercher le motif m très souvent, dans plusieurs textes différents, il suffit d'effectuer le prétraitement consistant à calculer ces tables une seule fois.

C Algorithme de Rabin-Karp

L'algorithme de Rabin-Karp est à nouveau une modification de l'algorithme naïf, dans lequel i augmente de 1 en 1, mais où l'on accélère les vérifications à l'aide d'une fonction de hachage. On calcule la valeur de hachage du motif, puis, à chaque position de départ i possible, on calcule la valeur de hachage de la fenêtre du texte concernée, et on ne compare le motif avec la fenêtre que si les valeurs de hachage sont identiques.

Afin que cette méthode soit utile, il faut que le calcul du hash d'une fenêtre soit plus rapide que la comparaison avec le motif. Pour cela, on utilise une fonction de hachage spéciale, telle que l'on peut calculer le hash d'une fenêtre à partir du hash de la fenêtre précédente en $\mathcal{O}(1)$.

Voyons un exemple de fonction de hachage garantissant cette propriété. On identifie Σ à $[\![0, |\Sigma| - 1]\!]$: lettre est considérée comme un nombre. On regarde ensuite la fonction de hachage suivante :

$$h : \begin{array}{ccc} \Sigma^* & \longrightarrow & [\![0, q-1]\!] \\ w & \longmapsto & \displaystyle\sum_{i=0}^{|w|-1} w_i |\Sigma|^{|w|-i-1} \mod q \end{array}$$

Autrement dit, $h(w)$ est w considéré comme un nombre en base $|\Sigma|$, puis pris modulo q. Par exemple, si l'on prend $\Sigma = \{A, B, C, D\}$, on a donc $|\Sigma| = 4$, et on prendra donc $A = 0, B = 1, C = 2, D = 3$. Pour $q = 10$, le hash du mot `BACBD` est $1 \times 4^4 + 0 \times 4^3 + 2 \times 4^2 + 1 \times 4 + 3 = 256 + 32 + 4 + 3 = 295 = 5 \mod 10$.

On remarque que l'on peut facilement calculer le hash d'une fenêtre à partir de celui de la fenêtre précédente. Considérons le cas général : un texte t de taille n, et un motif m de taille p. On notera $B = |\Sigma|$. Si l'on note $h_i = h(t_i \dots t_{i+p-1})$ le hash de la fenêtre de position i pour $i \in [\![0, n-p]\!]$, alors on a :

$$h_i = \sum_{k=0}^{p-1} t_{i+k} |\Sigma|^{p-k-1} \mod q$$

Donc, pour $i \in [\![0, n-p-1]\!]$:

$$\begin{aligned} h_{i+1} &= \sum_{k=0}^{p-1} t_{i+k+1} B^{p-k-1} \mod q \\ &= \sum_{l=1}^{p} t_{i+l} B^{p-l} \mod q \\ &= B \sum_{l=0}^{p-1} t_{i+l} B^{p-l-1} + t_{i+p} - t_i B^p \mod q \\ &= B h_i + t_{i+p} - t_i B^p \mod q \end{aligned}$$

Exemple 2. Avec le même alphabet et la même fonction de hachage, on considère le mot $t =$ `BADADBAC.` , calculons la valeur des hash des fenêtres de taille 3 de t (commencez par calculer $(-4^3) \mod 10$:

w		$h(w)$
`BAD`	$\mapsto$	
`ADA`	$\mapsto$	
`DAD`	$\mapsto$	
`ADB`	$\mapsto$	
`DBA`	$\mapsto$	
`BAC`	$\mapsto$	

Ainsi, on peut calculer h_{i+1} en fonction de h_i en temps $\mathcal{O}(1)$. L'algorithme de Rabin-Karp repose sur ce principe :

Algorithme 6 : Recherche de motif : Rabin-Karp

Entrée(s) : $t, m \in \Sigma^*$ avec $|t| = n$, $|m| = p$
Sortie(s) : i tel que m apparaît à la position i de t

```
1  h_t ← 0 // hash de la fenêtre du texte
2  h_m ← 0 // hash du motif
3  B ← |Σ| ;
4  r ← B^p mod q ;
5  pour j = 0 à p − 1 faire
6      h_t ← B h_t + t_j mod q;
7      h_m ← B h_m + m_j mod q;
8  i ← 0;
9  tant que i < n − p faire
10     si h_t = h_m alors
11         j ← 0 tant que j < p et t_{i+j} = m_j faire
12             j ← j + 1;
13         si j = p alors
14             retourner i
15     h_t ← B h_t − r t_i + t_{i+p} mod q;
16     i ← i + 1;
17 retourner Pas d'occurence
```

Complexité Dans le pire cas, la fonction de hachage cause de nombreuses collisions, et chaque hash de fenêtre calculé est égal au hash du motif, sans pour autant que la fenêtre soit égale au motif. Alors, l'algorithme ne va jamais profiter de l'accélération due au hachage, et s'exécuter en $\mathcal{O}(np)$. En pratique, l'algorithme s'effectue en temps linéaire en moyenne. Le choix de q impacte fortement les performances, et comme souvent pour des fonctions de hachage, les grands nombres premiers sont à privilégier.

2 Compression

Un algorithme de compression permet de réduire la taille d'un fichier, en l'encodant d'une certaine manière. Il va de pair avec un algorithme de décompression, qui doit connaître la méthode de compression utilisée pour pouvoir reconstruire l'information. On peut classifier les algorithmes de compression en deux familles : avec pertes (fichiers mp3 et mp4 par exemple) et sans pertes. Pour de nombreuses applications, les pertes ne sont pas acceptables : si l'on comprime un fichier C, on veut pouvoir le restituer à l'identique, au caractère près. On étudie dans cette section deux algorithmes classiques de compression sans pertes : l'algorithme de Huffman et l'algorithme de Lempel-Ziv-Welch (ou LZW).

A Introduction à la compression

On considère un alphabet Σ fini. Un encodage sur Σ^* est un couple de fonctions (c, d) avec $c : \Sigma^* \to \{0,1\}^*$ et $d : \{0,1\}^* \to \Sigma^*$ telles que pour tout texte $t \in \Sigma^*$, on a $d(c(t)) = t$.
On appelle c la fonction d'encodage et d la fonction de décodage.
Pour un texte $t \in \Sigma^*$, on appellera t le texte source et $c(t)$ le texte encodé.

Une méthode générale consiste à fixer pour chaque lettre $a \in \Sigma$ un code $c(a) \in \{0,1\}^*$, puis à encoder un mot $t = t_0 \dots t_{n-1} \in \Sigma^*$ par concaténation des codes des lettre :

$$c(t_0 \dots t_{n-1}) = c(t_0) \dots c(t_{n-1})$$

Si l'encodage choisi est tel que chaque lettre $a \in \Sigma$ a un code $c(a)$ d'une même longueur p, il est facile de décoder un texte encodé : il suffit de le lire par blocs de p bits et d'associer à chaque bloc la lettre correspondante.
Par exemple, l'encodage ASCII représente chaque caractère sur 8 bits. Le texte `Bonjour !!` correspondrait à la suite de 10 octets `42 6F 6E 6A 6F 75 72 20 21 21`, soit 80 bits.

On peut imaginer des codes où les lettres ne sont pas toutes encodées sur le même nombre de bits. Si l'on veut minimiser la taille du texte encodé, on pourrait essayer d'encoder les lettres les plus fréquences sur moins de bits.

Par exemple, on considère l'alphabet $\Sigma = \{a, b, c\}$, et le texte source $t =$ `aabaac`. Cet alphabet est de taille $3 \leq 2^2$, donc on pourrait utiliser un code de taille fixe sur 2 bits et encoder le texte avec 12 bits au total.
La lettre `a` apparaissant plus souvent que les autres, on peut aussi essayer l'encodage suivant : $a \mapsto 0$, $b \mapsto 01$ et $c \mapsto 10$. Alors on peut encoder le texte en 8 bits : 00010010. Cependant, **on ne peut plus décoder le texte**, car la fonction d'encodage n'est plus injective : les textes `ac` et `ba` s'encodent tous les deux en 010.
Deuxième essai : $a \mapsto 0$, $b \mapsto 10$ et $c \mapsto 11$. Alors, le texte précédent s'encode en 00100011 à nouveau en 8 bits, mais cette fois ci la fonction d'encodage est injective, on peut facilement décrire le procédé de décodage d'un texte codé s :

1. Si s est vide, renvoyer un texte vide
2. Si $s = 0s'$, renvoyer a suivi du décodage de s'
3. Si $s = 1xs'$ avec $x \in \{0,1\}$:
   (a) Si $x = 0$, renvoyer b suivi du décodage de s'
   (b) Si $x = 0$, renvoyer c suivi du décodage de s'

Cet algorithme fonctionne car il n'y a jamais de situation où l'on peut avoir lu une portion qui correspond à l'encodage d'un caractère ET qui pourrait être étendu en l'encodage d'un

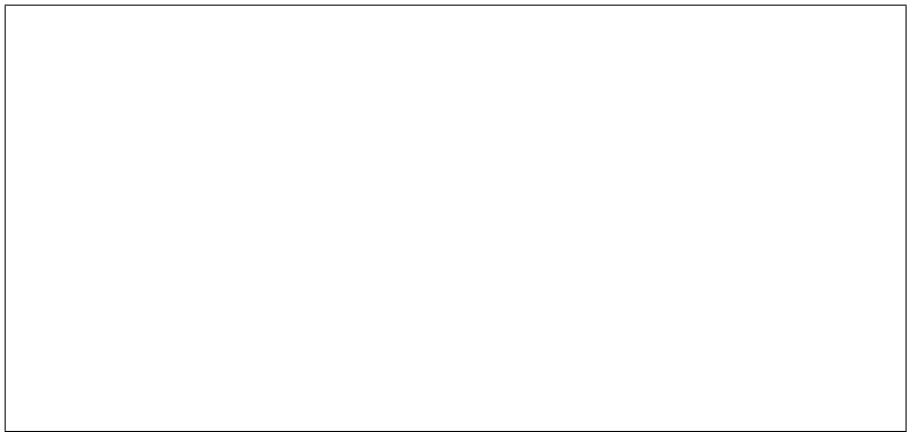
autre caractère. Au contraire, dans notre premier essai, lorsqu'on a lu un 0, on ne sait pas si l'on a fini de lire un a ou si l'on est en train de lire un b.

Définition 4. Un ensemble de mots $L \subseteq \Sigma^*$ est dit sans-préfixe (ou **prefix-free**) s'il n'existe pas deux mots $u \neq v \in L$ avec u préfixe de v.

Exemple 3. L'ensemble $\{0, 10, 1110, 1101\}$ est sans-préfixe.

Une fonction $c : \Sigma \to \{0,1\}^*$ dont l'image est sans préfixe peut être représentée par un arbre binaire dont les feuilles sont les lettres, et tel que pour une lettre $x \in \sigma$ donnée, le chemin de la racine jusqu'à x forme un mot binaire correspondant à $c(x)$.
Par exemple :

$$\begin{aligned} x &\mapsto 000 \\ p &\mapsto 0010 \\ m &\mapsto 0011 \\ t &\mapsto 0100 \\ i &\mapsto 0101 \\ d &\mapsto 011 \\ u &\mapsto 100 \\ e &\mapsto 101 \\ o &\mapsto 1100 \\ s &\mapsto 1101 \\ (espace) &\mapsto 111 \end{aligned}$$



Proposition 3. On considère une fonction d'encodage $c : \Sigma^* \to \{0,1\}^*$ telle que pour tout mot $t = t_0 \dots t_{n-1} \in \Sigma^*$, on a $c(t_0 \dots t_{n-1}) = c(t_0) \dots c(t_{n-1})$. On suppose que l'ensemble $\{c(a) \mid a \in \Sigma\}$ est sans-préfixe. Alors, c est injective, et peut donc être décodée.

Démonstration. On note A_c l'arbre binaire construit à partir de c sur les lettres de Σ. Étant donnée $B = b_0 \dots b_{m-1}$ une suite de bits (un texte encodé), on peut facilement retrouver son texte source à l'aide de A_c, il suffit d'utiliser B pour se déplacer dans l'arbre, en revenant à la racine à chaque fois que l'on arrive sur une feuille :

Algorithme 7 : decodage_arbre

Entrée(s) : $B = b_0 \dots b_{m-1}$ suite de bits, A_c arbre binaire de feuilles étiquetées par Σ
Sortie(s) : Texte $T = t_0 \dots t_{n-1}$ décodé depuis B selon l'encodage c

```
si B est vide alors
    retourner ϵ// mot vide
r ← racine de A_c;
i ← 0;
tant que r n'est pas une feuille faire
    Si i ≥ n, erreur d'encodage ;
    r ← enfant de r correspondant à b_i // 0 pour gauche, 1 pour droite
    i ← i + 1;
a ← lettre de r // r est une feuille
retourner a.decodage_arbre(b_i ··· _{n-1})
```

□

Exemple 4. Avec l'encodage dont on a dessiné l'arbre plus haut, décoder le message suivant :

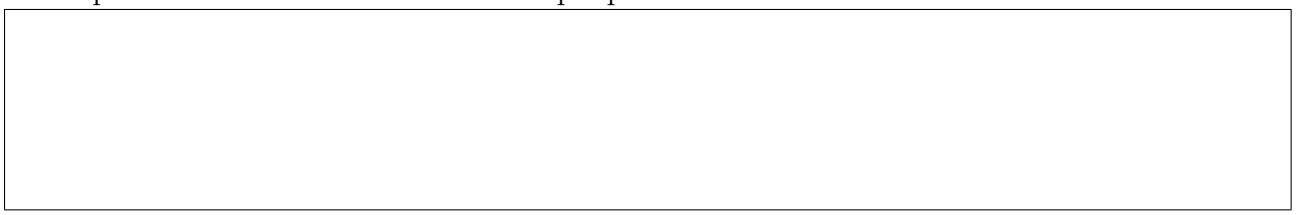
```
00110010 11101110 11000001 11010011 00100110
11110111 01110111 10110101 10110000 0
```

Sur l'exemple précédent, le texte initial fait 22 lettres. Comme il utilise moins de 16 caractères distincts, on aurait pu l'encoder de manière plus classique avec un encodage de taille fixe (4 bits par caractère), ce qui donnerait 88 bits au total. Avec l'encodage de taille variable proposé, on n'utilise que 73 bits.
Cependant, pour décoder le texte, il faut également connaître la valeur de l'encodage de chaque caractère, i.e. l'arbre binaire utilisé pour l'encodage. Pour des petits textes il sera plus judicieux de garder un encodage naïf comme le code ASCII, mais pour de très grands textes, le coût de stockage de l'arbre est largement compensé par la réduction de la taille du texte encodé, dès lors que l'encodage est bien choisi.

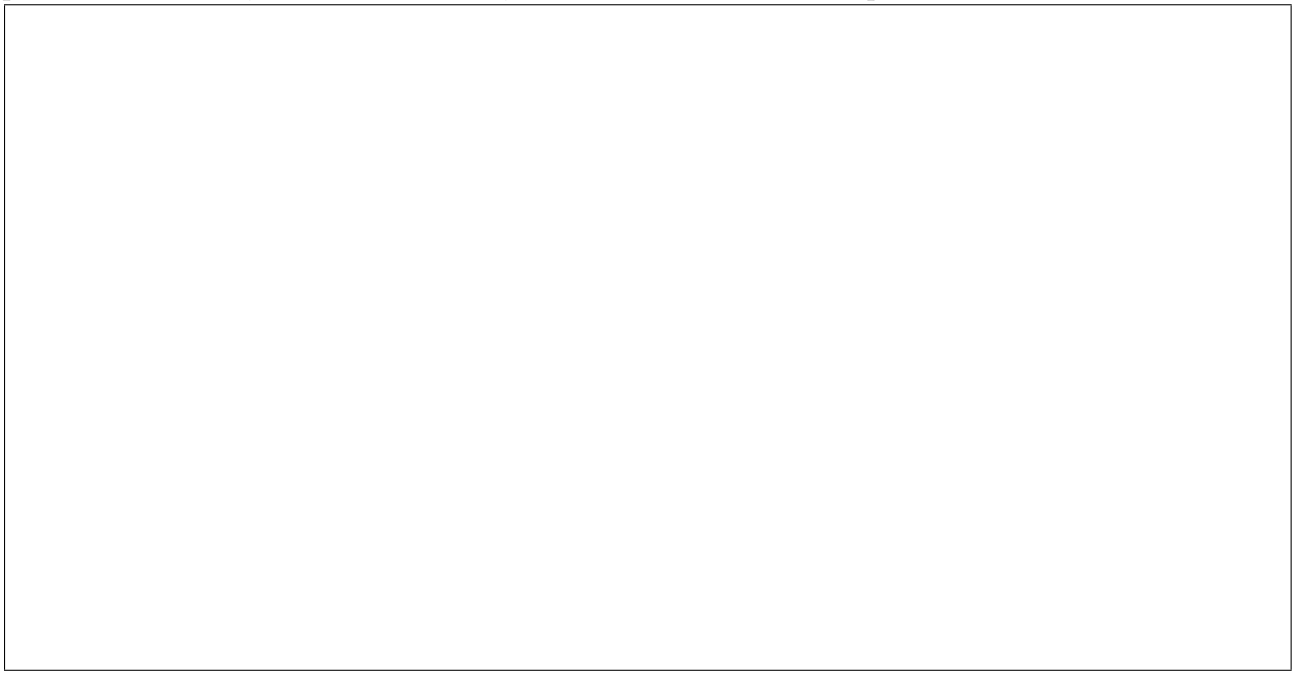
B Codage de Huffman

Le codage de Huffman consiste à trouver un arbre binaire donnant lieu à un encodage de taille minimale pour un texte donné, en encodant les lettres les plus fréquentes avec peu de bits, et les lettres les moins fréquentes avec plus de bits.

On considère un texte t sur l'alphabet Σ. L'algorithme de Huffman manipule des arbres binaires, dont les feuilles sont étiquetés par les lettres de Σ, et dont tous les nœuds sont étiquetés par une fréquence, le principe étant que la fréquence d'un nœud donne la somme des fréquences des feuilles qu'il enfante. Ainsi, un nœud de faible fréquence contiendra des lettres peu utilisées, et pourra être mis dans les profondeurs de l'arbre à moindre coût. A l'inverse, un nœud de forte fréquence contiendra des lettres très fréquentes, et doit être réservé afin d'être mis proche de la racine.
Au départ on dispose d'une liste de feuilles : une feuille par lettre a de Σ, étiquetée par a et par la fréquence de a dans t. Faisons l'exemple pour le texte $t =$ `TARTES` :



Puis, tant qu'il reste au moins deux arbres dans la liste, on extrait les deux arbres de fréquences minimales, et on les fusionne, en additionnant leurs fréquences :



L'arbre obtenu à la fin du processus est l'arbre de Huffman du texte. Voyons le pseudo-code (on notera $A.f$ la fréquence étiquetant la racine d'un arbre A) :

Algorithme 8 : Arbre de Huffman

Entrée(s) : $t \in \Sigma^*$ un texte source
Sortie(s) : A arbre de Huffman de t

$f \leftarrow$ dictionnaire des fréquences des lettres de Σ dans t;
$L \leftarrow$ liste vide;
pour $a \in \Sigma$ **faire**
    Ajouter à L l'arbre feuille $F(a, f[a])$;
tant que L ***contient au moins 2 arbres*** **faire**
    Sélectionner A_1 et A_2 dans L de fréquences minimales;
    Supprimer A_1 et A_2 de L et les remplacer par un noeud $N(A_1.f + A_2.f, A_1, A_2)$;
retourner ***l'unique élément de*** L

Une fois que cet arbre est créé, on peut encoder le texte source en remplaçant chaque lettre par son chemin dans l'arbre :

Algorithme 9 : Codage de Huffman

Entrée(s) : $t \in \Sigma^*$ un texte source
Sortie(s) : Suite de bits encodant t

$A \leftarrow$ arbre de Huffman de t;
$c \leftarrow$ dictionnaire associant à chaque $a \in \Sigma$ son chemin $c[a]$ dans l'arbre A;
$res \leftarrow []$;
pour a ***lettre de*** t **faire**
    Ajouter $c[a]$ à res;
retourner res

On peut montrer que pour un texte donné, le codage de Huffman est optimal parmi les codages lettre par lettre : il utilise le plus petit nombre de bits.

C Codage de Lempel-Ziv-Welch

L'algorithme de Huffman encode le texte lettre par lettre. Ainsi, même dans le meilleur des cas, il doit utiliser au moins un bit par lettre. On pourrait cependant imaginer des techniques d'encodage qui utilisent la structure du texte. Par exemple, si l'on veut décrire à quelqu'un le texte `ABC ABC ABC ... ABC ABC ABC` (avec 1000 occurrences), on peut lui dire "mille fois ABC", ce qui est bien plus efficace que d'essayer d'encoder le texte lettre par lettre.

L'algorithme de compression de Lempel-Ziv-Welch, ou algorithme LZW, consiste à encoder des parties du texte en faisant référence à des parties précédentes déjà encodées. Par exemple, si l'on a encodé un bloc du texte constitué des lettres `RADIS`, alors il sera moins cher d'encoder un bloc constitué des lettres `PARADIS`.

Plus précisément, l'algorithme de LZW prend en entrée un encodage initial des lettres Σ sur un nombre constant de bits (par exemple l'ASCII), et va construire un dictionnaire qui va associer à certains mots $u \in \Sigma^*$ un code. Initialement, ce dictionnaire ne contient que les lettres seules, et leur associe leur code selon l'encodage fourni. Puis, en lisant le texte à encoder, on rajoute au dictionnaire des blocs de texte de plus en plus gros, que l'on encode par des nouvelles valeurs.

Ce dictionnaire permettra d'encoder et de décoder le texte. Un avantage de l'algorithme LZW est que l'on peut reconstruire le dictionnaire à la volée en lisant le texte encodé. Il n'y a donc pas besoin de le transmettre en même temps que le texte. Il suffit de donner l'encodage initial pour les lettres, ce qui est généralement le cas, par exemple si l'on se met d'accord pour utiliser l'encodage ASCII. Regardons le pseudo-code de l'algorithme de LZW.

Algorithme 10 : Codage de LZW

Entrée(s) : t texte à encoder, D dictionnaire des codes des lettres
Sortie(s) : s suite de nombres encodant t

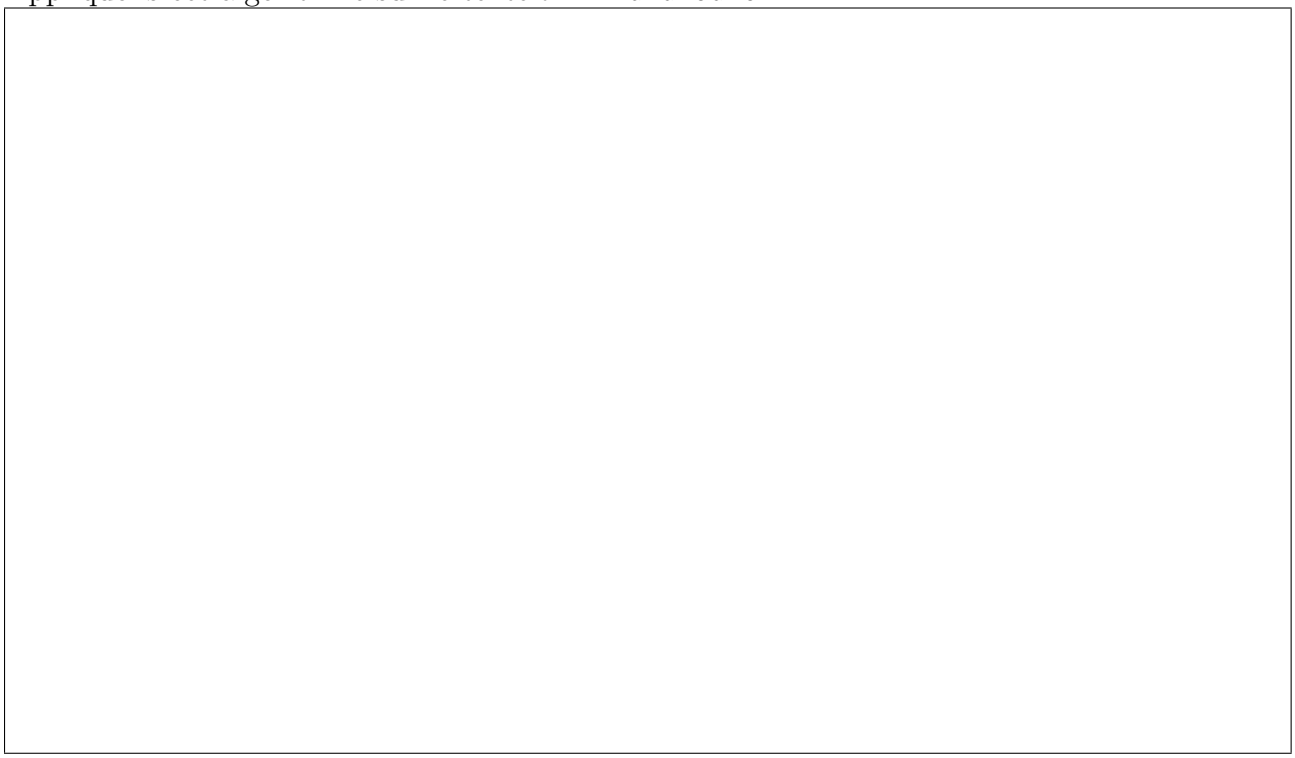
```
1  k ← code maximal utilisé dans D // 127 pour ASCII
2  k ← k + 1 // prochain code à utiliser
3  s ← [];
4  i ← 0// Indice pour parcourir t
5  w ← ""// Mot à rajouter au dictionnaire
6  tant que i < |t| faire
7      si w.t[i] n'est pas dans D alors
8          Ajouter D[w] à s;
9          D[w.t[i]] ← k;
10         k ← k + 1;
11         w ← t[i];
12     sinon
13         w ← w.t[i];
```

Un invariant de l'algorithme est : "s, w et $t[i..n[$ partitionnent le texte d'entrée". Plus précisément, à chaque instant, s est la partie du texte déjà lue, w est la partie en cours de lecture, et $t[i..n[$ la partie encore non-lue.

Exemple 5. On considère l'alphabet $\Sigma = \{A, C, G, T\}$ et comme encodage initial :

$$A \mapsto 0, C \mapsto 1, G \mapsto 2, T \mapsto 3$$

Ainsi, le premier code que l'on rajoutera au dictionnaire aura la valeur 4.
Appliquons cet algorithme sur le texte $t =$ `ATGAGACGACAT`



Une fois que l'on obtient la suite de nombres correspondant au codage LZW de t, il reste à transformer ces données en suite de bits. On peut par exemple écrire chaque nombre sur le même nombre p de bits (en prenant p le plus petit possible).

Exemple 6. Sur l'exemple précédent, sur combien de bits tient le code ? Combien de bits aurait-on utilisé pour écrire le texte de manière naïve ?

L'algorithme LZW donne de bonnes performances sur des textes longs, avec de nombreuses répétitions.

Exercice 5.

Question 1. Avec le même dictionnaire initial, appliquer LZW sur $t =$ `AAAAAAAAAAAAAAA` (15 'A'). Comparer le nombre de bits utilisés avec LZW et sans compression.

Question 2. Pour $n \in \mathbb{N}$, quel sera le code produit par l'algorithme de LZW sur le texte `AAAA...A` avec n `A` ? Quel sera le facteur de compression ?

Décodage