L'algorithmique du texte regroupe les différents problèmes que l'on peut étudier sur des suites de lettres, de mots. Quelques exemples de problèmes assez fondamentaux :

— La mesure de similarité entre deux textes ou fichiers. Résoudre ce problème permet de détecter le plagiat, de catégoriser des œuvres similaires, de corriger automatiquement les fautes d'orthographe.

— La recherche de motif. C'est un problème qui doit être résolu à chaque fois que vous utilisez le raccourci CTRL+F pour chercher un mot dans une page web ou dans un document.

— La compression de données. Comprimer un fichier permet par exemple de le transmettre à moindre coût, et de reconstruire le fichier d'origine (parfaitement ou non).

Nous avons déjà vu un exemple d'algorithme qui permet de mesurer la similarité entre deux textes : la distance de Levenshtein. Dans ce chapitre, on s'intéresse aux deux autres problèmes évoqués plus haut : la recherche de motif et la compression de données.

1 Recherche de motif

Le problème de la recherche de motif est de savoir, étant donné un texte t et un plus petit texte m (appelé motif), si m apparaît à une position de t.

Définition 1

Soit Σ un alphabet fini non vide. On rappelle que Σ^* désigne l'ensemble des suites finies de Σ, ce que l'on appelle aussi des **mots** sur Σ.

Soient $t = t_0 t_1 \dots t_{n-1}, m = m_0 \dots m_{p-1} \in \Sigma^*$ deux mots.

— La **fenêtre** de t de longueur k à la position i est le mot $t_i t_{i+1} \dots t_{i+k-1}$.

— On dit que m est un **motif** de t si m est égal à une des fenêtres de t de longueur p.

Par exemple, `LATIN` est un motif de `CHOCOLATINE`, car il est égal à la fenêtre de longueur 5 à la position 5. Formellement, le problème de recherche de motif est :

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 $t_i t_{i+1} \dots t_{t_i+p-1} = m$

Attention : dans la recherche de motif, on ne peut pas sauter de lettre : `CHINE` n'est pas un motif de `CHOCOLATINE` (on dit que c'est un *sous-mot*).

Exercice 1

Montrer que la recherche d'un sous-mot m de taille p dans un texte t de taille n peut s'effectuer en $\mathcal{O}(n+p)$.

De nombreuses variantes du problème de la recherche de motif sont envisageables : le premier indice d'une fenêtre correspondant au motif, la liste de tous les indices correspondant, etc... Certains algorithmes spécialisés permettent même de chercher un ensemble de motifs dans un texte.

A Première approche : force brute

L'algorithme le plus naturel pour résoudre de problème est de simplement tester toutes les fenêtres du texte, et de comparer chacune au motif lettre par lettre :

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

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

La complexité est en $\mathcal{O}(np)$ dans le pire cas. En pratique, l'algorithme peut s'exécuter plus rapidement, car la vérification d'une fenêtre s'arrête dès **qu'une** mauvaise lettre est trouvée.

Exercice 2

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 $\mathcal{O}(n-p)$.

On étudie deux algorithmes qui consistent en des améliorations assez directes de cet algorithme force-brute.

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 fenêtres plutôt que de les comparer lettre par lettre.

B Algorithme de Boyer-Moore-Horspool

Le premier algorithme que nous allons étudier nécessite de décaler le motif pour **aligner** certaines de ses lettres avec des lettres du texte.

Exercice 3

On suppose chercher un motif m dans un texte t. On suppose être en train de tester la position i du texte, et on note k, j deux positions du motif avec $k < j$:

t_0	…	t_i	…	t_{i+k}	…	t_{i+j}	…
		m_0	…	m_k	…	m_j	…

En face de quelle position i' du texte doit-on mettre m_0 pour aligner m_k avec t_{i+j} ?

t_0	…	t_i	…	$t_{i'}$	…	t_{i+j}	…
				m_0	…	m_k	…

Commençons par modifier l'algorithme naïf pour qu'il compare le motif et les fenêtres du texte en commençant par la **dernière** lettre du motif, mais en testant toujours les fenêtres dans l'ordre croissant. Prenons $t =$ `BABACACABAAAAC` et $m =$ `BAAAA`. Pour $i = 0$, on compare le motif `BAAAA` et la portion du texte `BABAC`.

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 voit que la dernière lettre ne correspond pas, le motif ne se trouve donc pas à la position $i = 0$. Dans l'algorithme naïf, on passerait donc à $i = 1$, mais on peut faire mieux : on voit que la lettre ayant posé problème dans le texte, C, n'apparaît pas du tout dans le motif m. Inutile de tester les positions $i = 1, 2, 3, 4$: elles mettraient le C en face d'un caractère du motif qui ne peut pas correspondre.

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 immédiatement passer i à 5 et recommencer. On compare donc le motif `BAAAA` et la portion du texte `ACABA`.

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				

La dernière lettre correspond, mais pas l'avant dernière : A dans m contre B dans t. Le motif contient un B, donc on ne peut pas appliquer directement le raisonnement précédent, mais 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 $i = 5, 6, 7$, qui mettraient le B du motif en face d'une lettre différente dans t :

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 $i = 8$, qui est la première position mettant un B du motif en face du B du texte. En comparant lettre par lettre, on trouve bien le motif à cet indice :

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	

La règle du mauvais caractère est une généralisation des remarques précédentes.

Définition 2

Soit $m \in \Sigma^*$ un motif de taille p et $a \in \Sigma$. La dernière occurrence 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 $\max \emptyset = -1$).
Autrement dit, $d_m(a)$ est le dernier indice où a apparaît dans m, ou bien l'avant dernier si m se termine par a.

Exemple 1

Si $m = BAACBEC$, avec $\Sigma = \{A, B, C, D, E\}$, alors on a les valeurs de d_m suivantes :

a	A	B	C	D	E
$d_m(a)$	2	4	3	-1	5

On remarque que pour une lettre a, a n'apparaît pas dans $m_{d_m(a)} \dots m_{p-2}$. Éventuellement, a peut apparaître en m_{p-1}.

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.

La règle du mauvais caractère permet donc de faire directement l'indice de recherche de i à $i + j - d_m(t_{i+1})$.

Exemple 2

Reprenons $m = BAACBEC$. On a :

a	A	B	C	D	E
$d_m(a)$	2	4	3	-1	5

Si l'on cherche m dans `AECDACCECD` :

A	E	C	D	A	C	C	E	C	D
B	A	A	C	B	E	C			

Avec $i = 0$ et $j = 5$, on a $m_j = E \neq C = t_{i+j}$. On cherche donc dans m la dernière occurrence non finale de la lettre C, qui vaut 3. On décale donc pour mettre m_3 en face du C problématique :

A	E	C	D	A	C	C	E	C	D
		B	A	A	C	B	E	C	

Ceci revient à avoir décalé l'indice de recherche i de 0 à $0 + 5 - 3 = 2$. On voit pourquoi on calcule la dernière occurrence **non finale** : si on avait pris en compte le C à la fin du motif, la règle du mauvais caractère nous aurait dit de reculer !

Démonstration. Montrons la règle du mauvais caractère.
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 3 : 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 Plutôt que de calculer $d_m(t_{i+j})$ à chaque fois que l'on en a besoin dans l'algorithme, on **pré-calcule** au préalable toutes les valeurs de d_m, que l'on stocke dans un tableau. Cette construction initiale est coûteuse, mais fait que l'on peut accéder à n'importe quelle valeur de $d_m(a)$ en $\mathcal{O}(1)$.

Exercice 4

Q1. On considère le motif $m =$ `BIBIDIBABIDI`. Calculer la table de d_m avec comme alphabet $\{A, B, C, D, I\}$.

Q2. Proposer un algorithme prenant en entrée un motif m, et qui précalcule les valeurs de d_m et les stocke dans un tableau. On supposera que les lettres de l'alphabet Σ sont numérotées : $\Sigma = \{c_0, \ldots, c_{|\Sigma|-1}\}$. Par exemple, pour un texte ASCII, on construirait un tableau de 128 caractères. Complexité attendue : $\mathcal{O}(p)$ avec $p = |m|$.

Algorithme 4 : `pre_calcul_MC`(m)

Entrée(s) : $m = m_0 \ldots m_{p-1}$ un motif
Sortie(s) : D tableau avec $D[i] = d_m(c_i)$ pour $0 \leq i < |\Sigma|$

Exercice 5

Comparons les performances de l'algorithme BMH avec celles l'algorithme naïf. Nous allons utiliser comme métrique le nombre de **comparaisons de lettres** effectuées par les deux algorithmes. On compte donc le nombre de fois que l'on teste l'égalité entre deux lettres.

On considère le texte t et le motif m suivants :

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

Appliquer l'algorithme naïf, et compter le nombre total de tests d'égalité de lettres, puis faire la même chose avec l'algorithme de BMH.

Complexité Dans tous les cas, le pré-traitement demande un temps $\mathcal{O}(p)$ additionnel. Pour la phase de recherche, dans le pire cas, la règle du mauvais caractère peut ne rien accélérer, et l'algorithme reste alors en $\mathcal{O}(np)$. Cependant, si l'on a de la chance, la règle permet de sauter l'intégralité de chaque fenêtre, et la phase de recherche prend un temps $\mathcal{O}(\frac{n}{p})$.

Exercice 6

Donner un cas général (avec n et p quelconques) où l'exécution prend un temps $\Theta(np)$, et un autre où elle prend $\mathcal{O}(\frac{n}{p})$ (sans compter le prétraitement).

C Règle du bon suffixe

L'algorithme de Boyer-Moore complet utilise une deuxième règle permettant d'accélérer encore plus la recherche : la règle du **bon suffixe**.

Définition 3

Soit $m = m_0 m_1 \dots m_{p-1} \in \Sigma^*$.

— Le mot $m_0 \dots m_{i-1}$ est le **préfixe** de taille i de m.

— Le mot $m_{p-i} \dots m_{p-1}$ est le **suffixe** de taille i de m.

Par exemple, ABA est le préfixe de taille 3 de $ABACAAB$, tandis que CAB est son suffixe de taille 3.

La règle du bon suffixe va consister à étudier les occurrences des différents suffixes du motif à l'intérieur de lui-même. Voyons un exemple : on cherche le motif $m =$ `CBACABACBA` dans le texte $t =$ `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`, coïncident, mais pas la troisième : 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 déjà 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 $i = 3$: le suffixe `ABACBA` du motif correspond, mais pas le C précédent, qui est en face d'un B dans le texte. Il n'y a pas d'autre occurrence de `ABACBA` 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 faire correspondre `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 4

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 aussi suffixe de $m_j \dots m_{p-1}$. Cas particulier : on pose $p_m(0) = p_m(1)$ pour empêcher de considérer m tout entier.

Exercice 7

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
```

Exercice 8

Appliquer l'algorithme pour $m =$ `BABABCADABAB` et le texte t suivant :

$$t = \texttt{AABCCBABABCADABABADDABBABABCADABABABCADABABCBAD}$$

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 $\mathcal{O}(np)$, ce qui n'est pas mieux que l'algorithme naïf ! Cependant, on peut montrer que pour trouver la première occurrence, la partie recherche s'exécute en $\mathcal{O}(n+p)$, même dans le pire cas.

Remarquons que les tables D, P et S ne dépendent que du motif m. Si l'on veut chercher un motif m souvent, dans des textes différents, on n'a besoin de pré-calculer les tables qu'une seule fois.

D Algorithme de Rabin-Karp

L'algorithme de Rabin-Karp consiste à calculer, pour chaque fenêtre, une valeur de hachage. A chaque étape, on commence par comparer le hash de la fenêtre avec celui du motif : s'ils ne coïncident pas, on passe immédiatement à la fenêtre suivante. Sinon, on teste naïvement la fenêtre comme dans l'algorithme force-brute. Supposons que l'on dispose d'une fonction de hachage $h : \Sigma^* \to \mathbb{N}$.

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_m ← h(m) // hash du motif
2  pour i = 0 à n − p faire
3      h_t ← h(t_i ... t_{i+p−1}) // hash de la fenêtre
4      si h_t ≠ h_m alors
5          Aller directement au passage de boucle suivant;
6      j ← 0;
7      tant que j < p et t_{i+j} = m_j faire
8          j ← j + 1;
9      si j = p alors
10         retourner i
11 retourner Pas d'occurrence
```

L'efficacité de cette méthode dépend de la fonction de hachage choisie. Si l'on prend une fonction de hachage quelconque, on peut s'attendre à ce que son calcul utilise toutes les lettres du mot haché : le calcul de h_t à chaque tour de boucle prend donc $\mathcal{O}(p)$ et l'algorithme reste en $\mathcal{O}(np)$. Cependant, on peut fabriquer des fonctions de hachage coulissantes (en anglais : *rolling hashes*), qui permettent de calculer le hash d'une fenêtre en fonction du hash de la fenêtre précédente, en $\mathcal{O}(1)$.

On identifie Σ à $[\![0, |\Sigma|-1]\!]$: une lettre est considérée comme un chiffre, et on peut voir tout mot $w \in \Sigma^*$ comme un nombre écrit en base $|\Sigma|$. Un exemple simple de fonction garantissant cette propriété est de calculer la somme des lettres du mot haché :

$$h(w_0 w_1 \dots w_{p-1}) = w_0 + w_1 + \dots + w_{p-1}$$

Pour un texte t, si l'on connaît le hash d'une fenêtre i (notons le h_i) :

$$h_i = h(t_i t_{i+1} \dots t_{i+p-1} = t_i + t_{i+1} + \dots + t_{i+p-1}$$

alors le hash de la fenêtre $i+1$ se déduit immédiatement :

$$h_{i+1} = h(t_{i+1} \dots t_{i+p} = h_i - t_i + t_{i+p}$$

En pratique, cette fonction de hachage serait très mauvaise, car elle a beaucoup de collisions : de nombreux mots auront la même somme des lettres.

Exercice 9

Voyons un exemple de fonction de hachage plus complexe ayant la même propriété :

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

Autrement dit, $h(w)$ est w, considéré comme un nombre en base $|\Sigma|$, pris modulo q.
Cette fonction, appelée "Empreinte de Rabin-Karp", est la fonction généralement utilisée dans l'algorithme de Rabin-Karp.

Par exemple, si l'on prend $\Sigma = \{A, B, C, D\}$, on a donc $|\Sigma| = 4$. Pour $q = 10$, le hash du mot `BACBD` est :

$h(BACBD) = 1\times4^4+0\times4^3+2\times4^2+1\times4+3 \bmod 10 = 256+32+4+3 \bmod 10 = 295 \bmod 10 = 5$

Considérons le cas général : un texte t de taille n et une taille de fenêtre p. On note $B = |\Sigma|$. 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 :

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

Q1. Exprimer h_{i+1} en fonction de h_i

Q2. En déduire le pseudo-code de l'algorithme de Rabin-Karp utilisant spécifiquement la fonction h définie dans cet exercice :

Algorithme 7 : `rabin_karp`(t, m)

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

Complexité Dans le pire cas, même avec une fonction de hachage parfaite, toutes les fenêtres peuvent être différentes du motif mais avoir le même hash. Dans ce cas extrême, l'algorithme ne profite jamais de l'amélioration apportée par le hachage, et l'algorithme va s'exécuter en $\mathcal{O}(np)$. Dans la majorité des cas cependant, l'algorithme va s'exécuter en temps linéaire si la fonction de hachage est efficace.

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, où le fichier obtenu après décompression n'est pas exactement le même que le fichier original. C'est le cas des fichiers mp3 (audio), mp4 (vidéo) ou jpeg (image).

— Sans pertes, où le fichier décompressé doit être identique au fichier original, au bit près.

Les algorithmes de compression avec perte sont utilisés lorsque la perte d'information ne nuit pas à l'utilisation du fichier. C'est le cas pour les vidéos ou la musique, du moment que les pertes sont mineures et n'affectent pas la qualité.
Pour de nombreuses applications en revanche, les pertes ne sont pas acceptables : si l'on comprime un fichier C, on veut pouvoir le restituer à l'identique, au caractère près.
Au programme, on s'intéresse à deux algorithmes classiques de compression sans pertes : l'algorithme de Huffman et l'algorithme de Lempel-Ziv-Welch (ou LZW).

A Définitions et exemples introductifs

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 simple 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 5

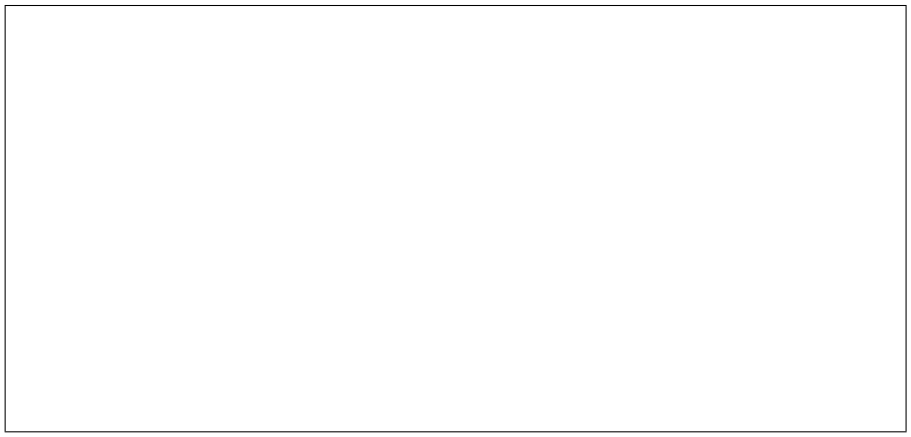
Un ensemble de mots $L \subseteq \Sigma^*$ est dit sans préfixe s'il n'existe pas deux mots $u \neq v \in L$ avec u préfixe de v.

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 $c(t_0 \dots t_{n-1}) = c(t_0) \dots c(t_{n-1})$, i.e. une fonction entièrement caractérisée par son action sur les lettres. 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 8 : 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 correspondant à un code préfixe optimal un texte donné, en encodant les lettres les plus fréquentes avec peu 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, de telle sorte que la fréquence d'un nœud est la somme des fréquences de ses feuilles. Un nœud de faible fréquence contient donc des lettres peu utilisées, et peut être mis dans les profondeurs de l'arbre. A l'inverse, un nœud avec une grande fréquence contient des lettres courantes, et doit être mis proche de la racine.
On part initialement d'une liste de feuilles : une feuille par lettre a de Σ, étiquetée par a et par la fréquence de a dans t. 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.
Faisons l'exemple pour le texte $t =$ `TARTES` :

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 9 : Arbre de Huffman

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

```
1 f ← dictionnaire des fréquences des lettres de Σ dans t;
2 L ← liste vide;
3 pour a ∈ Σ faire
4     Ajouter à L l'arbre feuille F(a, f[a]);
5 tant que L contient au moins 2 arbres faire
6     Sélectionner A_1 et A_2 dans L de fréquences minimales;
7     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);
8 retourner l'unique élément de L
```

On peut montrer que pour un texte donné, le codage de Huffman est optimal parmi tous 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 que tout le monde dispose de l'encodage initial. On peut alors soit utiliser un encodage classique comme l'ASCII, soit transmettre l'encodage initial avec le code encodé.

Regardons le pseudo-code de l'algorithme de LZW.

Algorithme 10 : `lzw`(t, D)

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 ← []// Résultat
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 10

Q1. Avec le même dictionnaire initial, appliquer LZW sur $t =$ `AAAAAAAAAAAAAAA` (15 'A'). Comparer le nombre de bits utilisés avec LZW et sans compression.

Q2. 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