

TP 9

Recherche dans un texte

Le but de ce TP est d'abord d'étudier des méthodes pour rechercher un mot dans un texte.

Rappelons qu'un texte est encodé par une chaîne de caractères s . Un mot à chercher sera donc lui aussi une chaîne m . Cependant nous prenons « mot » au sens général de « sous-chaîne » c'est à dire que les caractères de m se retrouvent les uns à la suite des autres tels quels dans s . Par exemple dans $s = \text{"abracadabra"}$ on trouve les mots "cad" qui apparaît à partir de la position 4 (la lettre c est en position 4 dans s , qui est numérotée à partir de 0) ou bien "abra" qui apparaît deux fois, à partir des positions 0 ainsi que 7.

Ainsi les programmes s'appliquent à rechercher un mot dans un texte en français (au sens habituel) mais aussi à rechercher un motif dans une séquence ADN. Par exemple dans la séquence $s = \text{"TTAATGCAATAAC"}$ on peut vouloir rechercher le motif "AAT" , qui apparaît deux fois, ou "ATT" qui n'apparaît pas.

Pour des mots en français il n'est pas très facile de distinguer si le mot apparaît bien de façon isolée ou non. Par exemple dans $s = \text{"j'aime travailler"}$ le mot "travail" apparaît bien, en tant que suite de lettres consécutives, et le mot "ai" apparaît deux fois. Distinguer les mots isolés serait nettement plus compliqué (il doit être précédé, et suivi, d'espaces, ou bien de virgules, de points, etc) donc nous ne nous préoccupons pas de ce problème dans ce TP.

Enfin, le TP vient avec un fichier joint `livre.txt` qui contient l'intégralité du livre *Le Rouge et le Noir* de Stendhal, permettant de faire des tests pour trouver des vrais mots.

I Fonctions préliminaires

Les fonctions suivantes ne seront pas utilisées telles quelles, mais constituent des révisions.

Exercice 1. Un pré-requis indispensable est de savoir chercher un caractère tout seul.

1. Écrire une fonction `cherche_caractere(s, x)` qui prend en argument une chaîne de caractères s et un caractère seul x , affiche tous les indices auxquels le caractère x apparaît dans s .
2. Améliorer la fonction pour écrire la fonction `compte_caractere(s, x)` qui renvoie le nombre d'apparitions du caractère x dans s .

On se donne maintenant une chaîne de caractère s et un mot m à chercher dedans. Remarquons déjà que la longueur de m est inférieure à celle de s , sinon le mot ne peut pas apparaître. . . On souhaite écrire une fonction `le_mot_est_ici(s, m, i)` qui prend en argument un indice i de la chaîne s , et qui va renvoyer `True` si le mot m est bien présent dans s à partir de cet indice, c'est à dire si $m[0]$ est égal à $s[i]$, et $m[1]$ est égal à $s[i+1]$, etc (on pourra consulter les illustrations de la section III.1). Attention car il y a deux chaînes, deux longueurs, il y a les indices dans s et les indices dans m , et il ne faut pas accéder à un indice d'une chaîne qui n'existe pas!

Exercice 2. Tout d'abord il y a une contrainte entre les longueurs des chaînes de caractères pour laquelle nous sommes sûrs que le mot n'est pas ici! Par exemple, un mot m de deux lettres ne peut pas démarrer sur le dernier indice de s . Un mot de trois lettres peut-il démarrer sur la dernière ou l'avant-dernière lettre de s ? Pouvez-vous donner la relation exacte entre les longueurs n de s , p de m et l'indice i ?

Exercice 3. Écrire la fonction `le_mot_est_ici(s, m, i)`.

Cela permet de répondre, déjà, à la question de recherche de mot.

- Exercice 4.**
1. En utilisant la fonction précédente, écrire une fonction `cherche_mot(s, m)` qui cherche à tous les indices possibles i de s si le mot m démarre bien à cet indice. La fonction affiche les indices i où on trouve le mot.
 2. Puis écrire une fonction `compte_mot(s, m)` qui compte combien de fois le mot apparaît.

Une variante intéressant est aussi d'afficher le mot trouvé *et un peu plus*, par exemple les 10 caractères précédents et 20 suivants. À cette fin et pour les parties suivantes, on rappelle les syntaxes de tranche : $m[a:b]$ sélectionne le mot extrait entre les indices a et b , $m[a:]$ sélectionne le mot extrait à partir de l'indice a et $m[:b]$ le mot jusqu'à l'indice b (exclus). On pourra écrire une fonction `cherche_mot_plus(s, m)` qui cherche le mot et affiche non pas les indices mais une tranche d'avant et après le mot.

II Avec des erreurs

On suppose maintenant que les mots peuvent contenir des erreurs d'orthographe, ou des simples variantes. Cela rend la recherche nettement plus compliquée. Étant donnée un mot s , on peut considérer qu'un autre mot t est à peu près égal à s si les deux diffèrent seulement sur une lettre, ou sur deux (ce qui suppose qu'ils ont même longueur), ou bien si l'un contient juste une lettre de plus que l'autre (insérée à un endroit quelconque), ou plusieurs, ou un mélange d'insertions et de substitutions.

On ne s'en sortira pas facilement si on ne suppose pas que les deux mots ont la même longueur et que les erreurs ne peuvent être qu'« alignées ». Par exemple les mots « PYTHON » et « PATRON » sont assez proches car seules les lettres Y avec A, et H avec R, doivent être échangées pour passer du premier au second ; mais avec « PIETON » l'écart porte sur trois lettres (et non pas deux) car le T n'est plus à la même place.

Définition 1. La **distance de Hamming** entre deux mots s et t **supposés de même longueur** est le nombre de lettres qui diffèrent à la même place, c'est à dire le nombre d'indices i tels que $s[i] \neq t[i]$.

Exercice 5. Écrire une fonction `distance(s, t)` qui calcule la distance de Hamming entre les deux mots s et t , supposés de même longueur (on pourra utiliser `assert` pour vérifier qu'ils le sont bien).

Pour rechercher un mot avec d'éventuelles erreurs, il est alors nécessaire de se fixer un seuil de tolérance auquel on considère qu'il s'agit du même mot : la distance de Hamming doit-elle être inférieure à 1 (au plus une lettre est autorisée à différer), ou bien 2, ou plus (on risque alors de trouver des mots franchement différents) ?

Exercice 6.

1. Écrire une fonction `le_mot_est_a_peu_pres_ici(s, m, i, seuil)` qui prend en argument une chaîne de caractères s , un mot à chercher m , un indice i et un seuil de tolérance qui est un nombre entier positif, et qui renvoie `True` si le mot dans s à partir de l'indice i est à une distance de m qui est inférieure ou égale au seuil.
2. En déduire une fonction `cherche_mot_erreurs(s, m, seuil)` qui cherche le mot m dans s avec le seuil de tolérance donné. Cette fois-ci, on aimerait vraiment que quand la fonction trouve à peu près le mot, elle affiche le mot trouvé (puisque ce ne sera pas exactement m mais des variantes), éventuellement avec quelques caractères avant et après.

III L'algorithme de Knuth-Morris-Pratt (KMP)

III.1 Quelques exemples

Exemple 1 Supposons qu'on cherche le mot $m = \text{"ATCGATG"}$ à l'intérieur de $s = \text{"ATCGATCGATCGATG"}$. L'algorithme le plus simple que nous avons étudié fonctionne comme une sorte de « fenêtre glissante » en tentant de faire correspondre le mot à chacune des positions possibles.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	T	C	G	A	T	C	G	A	T	C	G	A	T	G
A	T	C	G	A	T	G								
	A	T	C	G	A	T	G							
		A	T	C	G	A	T	G						
			A	T	C	G	A	T	G					
				A	T	C	G	A	T	G				
					A	T	C	G	A	T	G			
						A	T	C	G	A	T	G		
							A	T	C	G	A	T	G	
								A	T	C	G	A	T	G

(1)

On constate que le mot ne démarre pas en position 0, à cause de son G en position 6 qui ne coïncide pas avec le C de s .

Mais alors il est inutile de vouloir tester si le mot démarre en position 1 ou 2 ou 3. En effet, précisément le fait que l'échec ait lieu à cause de sa lettre G prouve que les lettres de s en position 1, 2 et 3 ne sont pas des A, et donc on peut directement vouloir tester si le mot démarre en position 4.

Encore une fois, à partir de la position 4 plusieurs lettres consécutives coïncident bien mais l'échec a lieu à cause du dernier G du mot, qui n'est pas égal au C de s en position 10. Et on remarque qu'on peut sauter directement à la position 8 pour tester si le mot est là. Ce qui sera bien le cas.

En résumé, en étudiant plus finement le mot m et la position à laquelle la comparaison échoue, on a réellement appelé la fonction `le_mot_est_ici` seulement 3 fois, et elle a réussi 1 fois.

Exemple 2 Étudions un cas plus simple où le mot à chercher m est constitué de lettres toutes différentes. Alors il est facile de se convaincre que si la fonction `le_mot_est_ici` échoue, on peut directement continuer à chercher dans s juste après le dernier échec de comparaison. Par exemple si on cherche le mot $m = \text{"AGCT"}$ dans $s = \text{"AGTAGCAGCT"}$ alors on le teste à partir de la position 0 (cela échoue sur le C de m), puis directement 2 (ce qui échoue directement sur le A) puis 3 (ce qui échoue à cause du T de m) puis enfin directement à 6. Résumons les tentatives effectuées par le tableau.

0	1	2	3	4	5	6	7	8	9
A	G	T	A	G	C	A	G	C	T
A	G	C	T						
		A	G	C	T				
			A	G	C	T			
						A	G	C	T

(2)

Exemple 3 La situation est moins simple quand une partie du mot à chercher se répète dans lui-même. Par exemple si on cherche $m = \text{"ACAT"}$ dans $s = \text{"ACACATAG"}$ alors on teste si le mot est en position 0, ce qui échoue à cause de son T (la comparaison en position 3 dans s) ; mais le mot démarre en fait à la position 2, donc on ne peut pas sauter directement à la position 3 ou après. En fait, on sait déjà que les lettres coïncident aux positions 2 et 3 et donc on peut poursuivre les comparaisons à partir de l'indice 4.

0	1	2	3	4	5	6	7
A	C	A	C	A	T	A	G
A	C	A	T				
		A	C	A	T		
				A	C	A	T

(3)

III.2 L'algorithme

L'idée de l'algorithme est d'abord de construire, étant donné le mot m , un plan qui puisse nous dire de combien d'indices sauter après avoir échoué à tester si le mot était à l'indice i dans s , en fonction du caractère rencontré dans s qui a fait rater la comparaison. Remarquons qu'en général c'est surtout la chaîne s qui est très grande, ainsi ce n'est pas forcément une contrainte lourde de faire des pré-calculs concernant le mot m seul pour pouvoir ensuite sauter plus rapidement.

Mais d'abord cela nécessite quelques définitions.

Définition 2. Étant donné un mot m :

1. Un **préfixe** de m est un sous-mot (c'est à dire, une suite de lettres consécutives de m) démarrant au début de m .
Par exemple **"GCC"** est un préfixe de **"GCCATC"**.
2. Un **suffixe** de m est un sous-mot terminant à la fin de m .
Par exemple **"ATC"** est un suffixe de **"GCCATC"**.
3. On convient que la chaîne vide **" "** est à la fois un préfixe, et un suffixe, de n'importe quel mot.
4. Le **bord** du mot m est le plus grand sous-mot (différent de m tout entier) qui est à la fois un préfixe et un suffixe.
Par exemple :
— **"AGT"** est le bord de **"AGTCGAGT"**,
— La chaîne vide **" "** est le bord de **"AGTCGACTC"**,

- "TTT" est le bord de "TTTGCCTTT", alors que "TT" ne l'est pas (c'est bien un préfixe et un suffixe mais il n'est pas le plus grand possible).

Exercice 7. 1. Écrire une fonction `est_bord(m, k)` qui renvoie `True` si les k premiers caractères du mot m sont aussi les k derniers — ainsi le bord de m est au moins de longueur k .

Attention à tous les indices manipulés et aux longueurs !

2. Écrire la fonction `longueur_bord(m)` qui renvoie la longueur du bord de m , en cherchant le plus grand k tel que la fonction précédente renvoie `True`.

Attention, elle peut très bien renvoyer `False` pour une valeur de k mais `True` pour des valeurs plus grandes ! Par exemple dans $m = \text{"AGTCGAGT"}$ la fonction précédente renvoie `True` seulement pour $k = 3$. Mais elle renvoie évidemment `True` pour $k = 0$. Il faut donc garder en mémoire le plus grand k qu'on a rencontré pour lequel on a obtenu `True`.

3. En déduire une fonction `longueurs_bords_prefixes` qui prend en argument une chaîne de caractères m et qui renvoie la liste B des longueurs du bord de chaque préfixe de m . Ainsi pour tout indice j , $B[j]$ sera la longueur du bord du préfixe formé des $j + 1$ premiers caractères de m , c'est à dire de $m[:j+1]$.

Par exemple, l'appel `longueurs_bords_prefixes("AATGAATC")` devra renvoyer la liste $[0, 1, 0, 0, 1, 2, 3, 0]$. En effet :

- "" est le bord de la chaîne "A",
- "A" est le bord de la chaîne "AA",
- "" est le bord des chaînes "AAT" et aussi "AATG",
- "A" est le bord de la chaîne "AATGA",
- "AA" est le bord de la chaîne "AATGAA",
- "AAT" est le bord de la chaîne "AATGAAT",
- "" est le bord de la chaîne "AATGAATC".

L'algorithme KMP fonctionne ainsi :

1. On se donne une chaîne s et un mot m à chercher, et on calcule la liste des longueurs des bords des préfixes B ci-dessus.
2. On initialise une variable i à 0, c'est un indice où chercher le mot dans s , et on démarre une boucle sur i . On préférera une boucle `while` plutôt que `for`, ce qui permet de faire varier les tailles du saut à l'indice suivant.
3. On cherche le plus petit indice j , à partir de 0 et s'il existe, pour lequel $s[i+j] \neq m[j]$.
 - S'il n'y en a pas, c'est que le mot m se trouve bien ici en démarrant à l'indice i .
 - Si $j = 0$, c'est que la comparaison rate sur la première lettre. On passe alors simplement à l'indice $i+1$.
 - En général, on saute à l'indice $i + j - B[j-1]$ de s , mais en démarrant la comparaison avec m à partir de son indice $B[j-1]$.

Exercice 8. Compléter le programme pour écrire la fonction `cherche_KMP(s, m)`.

Pour observer le fonctionnement du programme, on pourra rajouter des instructions `print` pour afficher les caractères comparés et le nombre de sauts effectués.

Exercice 9. Appliquer l'algorithme à la main sur l'exemple de la partie précédente.