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.

Dans tout le TP, un « texte » désigne une chaine de caractères quelconque qu'on notera `s`. Un « mot » désigne simplement une sous-chaine de caractères, c'est-à-dire une chaine `m` telle que les caractères consécutifs de `m` se retrouvent tels quels consécutivement dans `s`.

Par exemple dans `s = "abracadabra"` on trouve les mots `"cad"` qui apparait 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 apparait 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 = "TTAATGCAATAAC"` on peut vouloir rechercher le motif `"AAT"`, qui apparait deux fois, ou `"ATT"` qui n'apparait pas.

Le TP vient avec un fichier joint `livre.txt` qui contient l'intégralité du livre absolument passionnant *Le Rouge et le Noir* de Stendhal, permettant de faire des tests pour trouver des vrais mots. Ce livre étant tombé dans le « domaine public », chacun a le droit de le télécharger et de l'utiliser.

I L'algorithme simple

L'algorithme le plus simple que nous étudions consiste en une sorte de « fenêtre glissante » qui tente de faire correspondre le mot `m` à chacune des positions possibles dans le texte `s`.

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

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 chaine de caractères `s` et un caractère seul `x`, affiche tous les indices auxquels le caractère `x` apparait 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 chaine 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 apparaitre… On souhaite écrire une fonction `le_mot_est_ici(s, m, i)` qui prend en argument un indice `i` de la chaine `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. Dans notre exemple ci-dessus lors de la recherche de `m = "AAT"` dans `s = "TTAATGCAATAAC"`, le mot est présent aux positions 2 ainsi que 7, donc `le_mot_est_ici(s, m, i)` doit renvoyer `True` pour $i = 2$ et pour $i = 7$ et `False` sinon. Attention à ne pas se mélanger dans les indices !

Exercice 2
Tout d'abord il y a une contrainte entre les longueurs des chaines 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 apparait.

Une variante intéressant est aussi d'afficher le mot trouvé *et un peu plus*, par exemple les caractères précédents et suivants. À cette fin et pour les parties suivantes, on rappelle les syntaxes de tranche : `s[a:b]` sélectionne le mot extrait entre les indices `a` et `b` de la chaine de caractères `s`, `s[a:]` sélectionne le mot extrait à partir de l'indice `a` et `m[:b]` le mot jusqu'à l'indice `b` (exclus). Dans la fonction `cherche_mot(m)` on propose la syntaxe (où `i` est l'indice où le mot est trouvé, et `p` est la longueur de `m`) :

```
print(s[i-30:i+p+30].replace("\n", " "))
```

Le `"\n"` est un **caractère de saut de ligne** (*newline* en anglais), c'est un caractère à part entière d'une chaine de caractère ou d'un fichier texte (tout comme les lettres, les espaces et la ponctuation) dont le but est d'indiquer un saut de ligne. Le fichier joint contient de nombreux sauts de lignes qui coupent les phrases, ce qui est un peu ennuyeux pour notre programme.

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
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. Similairement à la fonction `le_mot_est_ici`, on doit cette fois écrire une fonction`le_mot_est_a_peu_pres_ici`, que nous abrégerons `lmeappi(s, m, i, seuil)`. Elle prend en argument, en plus de la chaine `s`, du mot à chercher `m`, et d'un indice `i` où démarrer la recherche, un seuil de tolérance qui est un nombre entier positif. Elle 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, et `False` sinon. Écrire cette fonction.

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 comme précédemment.

III L'algorithme de Knuth-Morris-Pratt (KMP) (d'après concours TB 2022)

Nous proposons d'étudier une méthode qui peut être sensiblement plus efficace pour rechercher un mot dans un texte, si on compte en terme de nombre de comparaisons de caractères à effectuer. L'idée est qu'on n'est pas obligé de recommencer à chaque fois avec la fonction `le_mot_est_ici` à l'indice $i+1$ après avoir testé à l'indice i : on peut se servir de l'information du nombre de comparaisons vérifiées par `le_mot_est_ici` pour décaler de plus d'un cran la prochaine recherche.

III.1 Quelques exemples

Exemple 1 Supposons que le mot à chercher `m` est constitué de lettres toutes différentes, par exemple on cherche le mot `m = "AGCT"` dans `s = "AGTAGCAGCT"`. Alors si la fonction `le_mot_est_ici(s, m, i)` échoue (renvoie `False`), on peut directement continuer à chercher dans `s` juste après le dernier échec de comparaison.

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

Ici on teste d'abord 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, où on trouve le mot.

On voit que la « fenêtre glissante » peut avancer parfois plus vite que dans l'algorithme naïf.

Exemple 2 La situation est moins simple quand une partie du mot à chercher se « répète dans lui-même ». Par exemple si on cherche `m = "ACAT"` dans `s = "ACACATAG"` :

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

On teste d'abord si le mot est en position 0, ce qui échoue à cause de son T (la comparaison en position 3 dans `s`) ; mais on ne va pas sauter directement à position 3 ou après car le mot démarre en fait à la position 2. En fait, une fois qu'on sait déjà que les lettres A coïncident à la position 2, on peut *poursuivre* les comparaisons à partir de la position 3 pour essayer de savoir si le mot démarre à la position 2. À la fin, le fait que la coïncidence ait lieu prouve aussi que les lettre A coïncident bien à la position 4, et donc qu'on peut tenter de poursuivre les comparaisons à partir de la position 5 pour savoir si le mot démarre à la position 4 ; ce n'est ici pas le cas.

Exemple 3 Pour la recherche du mot `m = "ATCGATG"` à l'intérieur de `s = "ATCGATCGATCGATG"`, on obtient les sauts suivants, ne trouvant pas le mot aux positions 0 ni 4, mais 8 :

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

III.2 Les notions

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 des caractères de `m` qui ont bien été comparés et de la connaissance de « comment les caractères comparés au début du mot se retrouvent à la fin du même mot ».

Dans la problématique de recherche d'une séquence ADN, c'est surtout la chaine `s` qui est très grande, et le mot `m` peut contenir beaucoup de répétitions dans lui-même, ainsi ce n'est pas une contrainte lourde de faire des pré-calculs concernant le mot `m` seul pour pouvoir ensuite sauter plus rapidement dans `s`.

Mais d'abord cela nécessite quelques définitions.

Définition

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 chaine 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 chaine 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 = "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(m)` 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 chaine `"A"`,
- `"A"` est le bord de la chaine `"AA"`,
- `""` est le bord des chaines `"AAT"` et aussi `"AATG"`,
- `"A"` est le bord de la chaine `"AATGA"`,
- `"AA"` est le bord de la chaine `"AATGAA"`,
- `"AAT"` est le bord de la chaine `"AATGAAT"`,
- `""` est le bord de la chaine `"AATGAATC"`.

III.3 L'algorithme

L'algorithme KMP fonctionne ainsi :

1. On se donne une chaine `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 les exemples de la partie III.1, et comparer avec les testes du programme.