TP 9
Texte et mots

Dans ce TP nous étudions trois problèmes autour de la recherche de mots dans un texte.

Précisons d'abord que durant 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.

Dans la partie I, on s'intéresse simplement à la recherche d'un mot dans un texte. On peut utiliser pour cela le fichier joint `livre.txt` qui contient l'intégralité du livre absolument passionnant *Le Rouge et le Noir* de Stendhal, pour y chercher et trouver des vrais mots. Ce livre étant tombé dans le « domaine public », chacun a le droit de le télécharger et de l'utiliser. Dans la partie II, on s'intéresse plutôt à la recherche d'un mot dans un dictionnaire, éventuellement avec l'objectif de corriger des erreurs d'orthographe. On peut alors utiliser le fichier joint `dictionnaire.txt` qui contient une liste de plus de 600 000 mots issue du logiciel libre de correction orthographique GNU Aspell. Enfin dans la partie III on étudie un algorithme un peu plus efficace pour la recherche de sous-séquences d'ADN.

I Recherche de mots dans un texte

Il s'agit dans cette partie de rechercher un mot, formé de plusieurs caractères consécutifs, dans un texte, éventuellement très long.

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

Exercice 1 *Mise en jambe*
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`.

L'algorithme 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	G	T	A	T	A	A	T	G	A	C
A	T	G										
	A	T	G									
		A	T	G								
			A	T	G							
				A	T	G						
					A	T	G					
						A	T	G				
							A	T	G			
								A	T	G		
									A	T	G	
										A	T	G

Sur cet exemple, on cherche le mot `m = "ATG"` dans `s = "TTAGTATAATGAC"` et on le trouve effectivement une fois, démarrant à la position 8 de `s`.

Chaque case remplie représente une comparaison effectuée entre une lettre de `m` et une lettre de `s`. Vert, la comparaison a réussi, et rouge, elle a échoué. Trois cases vertes consécutives et c'est le jackpot ! Dès qu'une case est rouge, alors on descend d'une ligne (on fait « glisser la fenêtre ») et on ré-essaie.

On étudie alors le cas général. On se donne 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 elle doit renvoyer `True` pour $i = 8$ uniquement et `False` sinon. Attention à ne pas se mélanger dans les indices !

Exercice 2

1. 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 ?
2. Écrire la fonction `le_mot_est_ici(s, m, i)`.

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

Exercice 3

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.

Pour l'utiliser sur l'exemple du livre, une variante intéressant est d'afficher le mot trouvé *et un petit peu peu plus*, par exemple une dizaine de caractères précédents et suivants. À cette fin et pour les parties suivantes, on rappelle la syntaxe de tranche : `s[a:b]` sélectionne le mot extrait entre les indices `a` et `b` de la chaine de caractères `s`. On propose d'insérer dans la fonction `cherche_mot(m)` le code (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 Correction orthographique

On souhaite maintenant chercher un mot dans un dictionnaire, éventuellement pour y corriger des erreurs d'orthographe.

Le fichier joint contient une liste énorme de mots (chaque mot de la langue française y apparait avec toutes ses variantes, chaque verbe avec toutes ses conjugaisons). Il est chargé par le fichier joint et la variable `dictionnaire` est alors une liste de mots. Dans un premier temps on peut utiliser l'opération `==` pour tester si deux mots sont égaux.

Exercice 4 *Mise en jambe*
Écrire une fonction `est_présent(m)` qui prend en argument un mot `m` et renvoie `True` si `m` est effectivement bien présent dans le dictionnaire, et `False` sinon.

Le but est maintenant d'étudier le problème de la correction orthographique, ce qui nécessite de définir à quelle condition on peut considérer que deux mots donnés sont « proches » (représentent le même mot mais l'un a potentiellement des erreurs d'orthographe). C'est un problème difficile en général… Mais nous allons nous restreindre à comparer uniquement des mots de même longueur.

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]`.

Par exemple, la distance de Hamming entre les mots `s = "vacances"` et `t = "savantes"` est 3. On peut le voir en essayant de superposer les deux mots :

0	1	2	3	4	5	6	7
V	A	C	A	N	C	E	S
S	A	V	A	N	T	E	S

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 exclure le cas où cette condition ne serait pas vérifiée).

Pour rechercher un mot dans un dictionnaire en tolérant d'éventuelles erreurs, on doit d'abord se fixer un *seuil*, c'est-à-dire décider combien d'erreurs au maximum on autorise. Avec un seuil de 0, on ne peut trouver que le mot lui-même. Mais si le seuil est trop gros, on risque de trouver beaucoup trop de mots (surtout si `m` est déjà court). Tester d'abord avec 1.

Exercice 6

Écrire une fonction `cherche_seuil(m, seuil)` qui cherche dans le dictionnaire, parmi tous les mots qui sont de même longueur que `m`, ceux qui ont une distance de Hamming inférieur à `seuil`, et les affiche.

En pratique, on ne sait pas à l'avance s'il y a bien des mots à distance 1 de `m`, ou bien si les mots les plus proches sont à distance 2, ou plus…

Exercice 7

Écrire une fonction `mots_les_plus_proches(m)` qui prend en argument un mot `m` (contenant éventuellement des erreurs d'orthographe) et qui calcule d'abord la distance minimale des mots, parmi tous ceux du dictionnaire, à `m` ; puis qui affiche tous les mots qui sont à cette distance minimale de `m` (cela nécessite donc d'itérer deux fois sur tout le dictionnaire).

Il existe un autre algorithme qui ne nécessite pas de calculer *d'abord* le minimum. On parcourt le dictionnaire en maintenant en mémoire à la fois une liste `L` des mots qu'on pense être les plus proches de `m`, et leur distance `dmin` à `m` :

- Si on lit un mot à distance plus grande que `dmin` : on ne fait rien et on passe à la suite.
- Si on lit un mot à distance exactement `dmin` : on l'ajoute à `L` (avec `append`).
- Si on lit un mot à distance inférieure à `dmin` : on met à jour la variable `dmin` (comme dans le calcul de minimum) et ausssi la liste `L`, qui devient égale au seul mot qu'on vient de lire. Donc on « jette à la poubelle » tous les mots accumulés jusque là, puisqu'on en a trouvé au moins un qui est encore plus proche !
- À la fin, `L` est bien la liste des mots les plus proches et `dmin` est la distance minimale d'un mot du dictionnaire à `m`.

Exercice 8 (*)

Écrire cette fonction `liste_mots_les_plus_proches(m)` qui renvoie la liste des mots les plus proches de `m`, en itérant une unique fois sur le dictinonaire, en fabriquant au fur et à mesure la liste des mots les plus proches quitte à vider la liste.

Les méthodes ci-dessus avec une simple petite variante permettent de trouver des mots pour compléter des grilles de mots fléchés ou des mots croisés. Dans ce contexte, on a des cases fixées et on doit chercher un mot rentrant dans ces cases, connaissant à l'avance le nombre de lettres ainsi que la position de certaines lettres. On utilisera le caractère `*` pour noter une case vide au contenu encore inconnu. Par exemple, la séquence `"p*t*on"` peut accepter le mot `"python"` mais aussi `"patron"` ou bien `"potion"`.

0	1	2	3	4	5
P	*****	**T**	*****	**O**	**N**
P	Y	T	H	O	N
P	A	T	R	O	N
P	O	T	I	O	N

Exercice 9

1. Écrire une fonction `est_acceptable(s, m)` qui prend en argument une chaine `s` éventuellement composée de caractères `*`, et un mot `m` (sans `*`, supposé de même longueur que `s`) et renvoie `True` s'il est possible de faire coïncider `m` sur `s`, et `False` sinon.
2. En déduire une fonction `complète(s)` qui prend en argument une chaine `s` comme ci-dessus et affiche tous les mots, parmi ceux du dictionnaire, qui peuvent coïncider sur `s`.

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

On reprend la situation de la partie I où le but est de rechercher un mot dans un texte. Nous proposons d'étudier une méthode qui peut être sensiblement plus efficace si on raisonne 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 (« décaler la fenêtre » d'un cran à chaque fois) : 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. La « fenêtre glissante » avance 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`.

Définition

Étant donné un mot `m` :

- 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"`.

- Un **suffixe** de `m` est un sous-mot terminant à la fin de `m`.

  Par exemple `"ATC"` est un suffixe de `"GCCATC"`.

- On convient que la chaine vide `""` est à la fois un préfixe, et un suffixe, de n'importe quel mot.

- 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 10

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 :

- 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.
- 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.
- 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 11
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 12
Appliquer l'algorithme à la main sur les exemples de la partie III.1, et comparer avec les tests du programme.