TP 8 : Recherche de motif dans un texte

Dans tout ce TP, nous nous donnons deux chaînes de caractères, `T` (qu'on appellera le texte) et `M` (qu'on appellera le motif). On notera |`T`| et |`M`| leurs longueurs respectives, et on pourra supposer que $|M| \leqslant |T|$.
Le but du TP est d'étudier différents algorithmes permettant de répondre à la question suivante : est-ce que le motif M apparaît dans le texte T (autrement dit, est-ce que `M` est une sous-chaîne de `T`) ?
Python possède une manière simple et élégante de répondre à cette question : il suffit d'utiliser `M in T`, qui renvoie `True` ou `False` suivant les cas.
On **s'interdira totalement l'utilisation de cette syntaxe** (non pas qu'elle soit moche ou inefficace, mais notre but ici est de comprendre ce qui peut se cacher derrière), et on ne s'autorisera que des comparaisons de deux caractères, et pas directement de deux chaînes.

Dans toute la suite, on utilise les mêmes notations qu'en Python : le premier caractère de la chaîne `T` est `T[0]`, et `T[ :k]` désigne la chaîne formée par les k premiers caractères de `T`, quand `T[i :k]` désigne la sous-chaîne de `T` formée des éléments d'indice entre i et $k-1$.
Une chaîne `a` est dite **préfixe** d'une chaîne `b` si `a = b[ :len(a)]` et `a` est dite **suffixe** de `b` si `a = b[len(b)-len(a) :]`.
Par exemple, si `T = champollion`, alors `champ` est un préfixe de `T`, `lion` en est un suffixe et `ampollio` n'est ni l'un ni l'autre.

Vous commencerez par télécharger le fichier `TP8.zip` *sur le site* `http ://www.mpsi2-champo.fr`*, puis par extraire tous les fichiers de l'archive dans votre dossier de travail.*
Ouvrez ensuite le fichier `TP8.py` *dans Pyzo, et commencez par compléter la première ligne avec le chemin de votre répertoire de travail, comme d'habitude, en remplaçant les* `\` *par des* `\\` *et en ajoutant un* `\\` *final.*

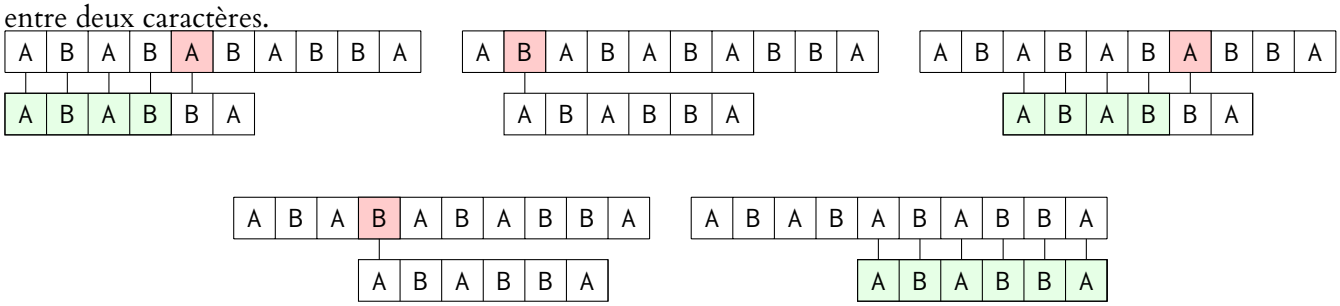
▶ **Question 0 (échauffement)** : écrire une fonction `prefixe(a,b)` qui prend comme paramètre deux chaînes de caractères et teste si `a` est un préfixe de `b`.

▶ L'algorithme naïf

La première méthode qui vient à l'esprit pour répondre à la question posée, est la suivante : vérifier si le premier caractère de `M` est égal au premier caractère de `T`.

- si c'est le cas, tester l'égalité des caractères en seconde position, etc.
- sinon, vérifier si le second caractère de `T` coïncide avec le premier caractère de `M` etc. Autrement dit, on cherche si `M` apparaît dans la chaîne `T[1 :]`.

Par exemple, les étapes de la recherche du motif `M = ABABBA` dans le texte `T = ABABABABBA` sont résumées par le dessin suivant, où chaque barre verticale reliant une lettre de `T` et une lettre de `M` indique une comparaison entre deux caractères.



▶ **Question 1** : écrire une fonction `naive(M,T)` (soit en partant de rien, soit en complétant le code partiel donné dans le fichier réponse) qui procède à cette recherche. Cette fonction devra retourner une liste, éventuellement vide, qui correspond aux indices des premières lettres des occurrences de `M`.
Par exemple, si `M = bonbon` *et que* `T = quelbonbonbon`*, alors la fonction* `naive` *doit retourner :* `[4,7]`*, car on trouve*

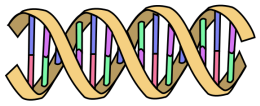
deux fois `M` *dans* `T` *: une fois à partir de* `T[4]` *et une fois à partir de* `T[7]`.

▶ **Question 2** : prouver que la complexité de la fonction précédente est en $O(|M| \times |T|)$. Quel est le cas le plus défavorable ?

▶ Quelques essais

Le principal inconvénient de la méthode naïve est que certains caractères de `T` sont lus plusieurs fois.
Ainsi, sur l'exemple illustré précédemment, c'est le cas du B qui est à la $4^{\text{ème}}$ lettre de `T`, qui est examiné 4 fois.
Malgré tout, en pratique, lorsqu'on cherche un mot dans un texte en français, le nombre de lettres lues plusieurs fois n'est pas si important.

▶ **Question 3** : modifier la fonction `naive` afin qu'un appel à cette dernière retourne le nombre exact de caractères de `T` qui ont été lus lors de son exécution.
Exécuter alors la cellule du fichier `TP8.py` qui commence par `QUESTION 3`. Une variable `nana` est alors automatiquement créée, qui est une chaîne de caractères contenant l'intégralité du roman *Nana* d'Émile Zola.
Déterminer alors le nombre de comparaisons de deux caractères que nécessite la recherche du motif «Berlin !» dans la variable `nana`. (*Vous noterez qu'il s'agit du dernier mot du roman.*)
Comparer ce nombre à la longueur totale du texte, et à la borne $|T| \times |M|$ obtenue ci-dessus.



La recherche de motif dans un texte est un thème central en bioinformatique, notamment pour l'analyse informatique de l'information génétique.
On rappelle que l'ADN est formé de 4 types de bases nucléiques : l'adénine (A), la cytosine (C), la guanine (G) et la thymine (T).
Une fois le génome d'un individu séquencé, pour déterminer s'il possède tel ou tel gène (par exemple pour déterminer si une souche de Covid est ou non le variant anglais dont on parle tant en ce moment), il faut déterminer si le génome complet (qui est donc une très longue chaîne formée des quatre lettres A,C,G et T) contient ou non une séquence donnée. Et donc il s'agit de chercher un motif dans un texte.

▶ **Question 4** : la fonction `ADNaleatoire(N)` fournie dans le fichier réponse permet de générer des chaînes de longueur `N` formées des 4 lettres A,C,G et T.
Générer un «grand génome» de 100 000 bases (ce qui est bien plus faible que les trois milliards de (paires de) bases d'un génome humain, et plus près de l'ordre de grandeur du génome d'un virus (un virus très en vue actuellement possède un génome de 30 000 paires de bases), plus adapté aux possibilités de notre machine), ainsi qu'un gène de quelques dizaines de bases.
Tester si ce gène appartient ou non au génome, et comparer le ratio $\dfrac{\text{nombre de comparaisons}}{|T|}$ avec celui de la question précédente.

▶ Utilisation d'une table des suffixes

Nous proposons ici une autre méthode pour chercher un motif dans un texte, bien plus rapide lors d'une recherche, mais qui nécessite d'effectuer au préalable un traitement du texte `T`.
On supposera à présent que la variable `T` est une variable que vous avez déjà définie dans votre code, et qui contient le texte dans lequel on souhaite effectuer les recherches. Vous pouvez par exemple prendre `quelbonbonbon`.
À présent, on cherche uniquement à déterminer si `M` apparait ou non dans `T`, et pas à déterminer toutes les positions auxquelles il apparaît.

▶ **Question 5 :** écrire une fonction `suffixe(i)` qui retourne `T[i :]`, le suffixe de `T` commençant à la $i^{\text{ème}}$ lettre ($0 \leqslant i <$ `len(T)`).

On souhaite alors créer une liste de tous les suffixes de `T`, triée par ordre croissant (pour l'ordre lexicographique). Une solution naïve serait d'utiliser

```
S = [suffixes(i) for i in range(len(T))]
S.sort()
```

Toutefois, cette solution est très coûteuse en espace, puisque les derniers suffixes ont une taille qui est très similaire à celle de `T`. Il est même facile de prouver que ceci nécessite une mémoire de l'ordre de $|T|^2$, ce qui va vite s'avérer catastrophique lorsque `T` est grand.
Nous proposons donc de trier plutôt les entiers de 0 à `len(T)-1`, suivant l'ordre lexicographique des suffixes. Sur la figure ci-dessous pour `T = 'quelbonbonbon'`, le tableau de gauche représente la liste des suffixes avant tri, la seconde représente la liste triée.

0	quelbonbonbon
1	uelbonbonbon
2	elbonbonbon
3	lbonbonbon
4	bonbonbon
5	onbonbon
6	nbonbon
7	bonbon
8	onbon
9	nbon
10	bon
11	on
12	n

10	bon
7	bonbon
4	bonbonbon
2	elbonbonbon
3	lbonbonbon
12	n
9	nbon
6	nbonbon
11	on
8	onbon
5	onbonbon
0	quelbonbonbon
1	uelbonbonbon

Plutôt que de garder en mémoire une liste complète de tous les suffixes, il nous suffit de conserver la liste $[10, 7, 4, 2, 3, 12, 9, 6, 11, 8, 5, 0, 1]$. Cette liste est dans la suite appelée **liste des suffixes de `T`**.

Il est possible en Python de trier une liste à l'aide d'une clé de tri, et pas forcément suivant les valeurs de la liste, mais plutôt sur les valeurs que prend une certaine fonction sur cette liste.
Par exemple, si on souhaite trier la liste `L = ['exponentielle','cosinus','sinus','racine']` par ordre lexicographique, il suffit d'utiliser `L.sort()`.
En revanche, si on souhaite trier ces mots par longueur croissante, on peut utiliser `L.sort(key = len)`, ce qui signifie que la fonction `len` est appliquée à chacune des quatre chaînes de caractère, puis que la liste est triée suivant ces longueurs. On obtient alors la liste `L = ['sinus','racine','cosinus','exponentielle']`, les mots étant rangés du plus court au plus long.

▶ **Question 6** : en utilisant ainsi la fonction `suffixe` comme clé de tri, former la liste (qu'on appelera `S` dans toute la suite) des suffixes de `T`. Vérifier que pour `T = 'quelbonbon'` vous obtenez bien la liste donnée ci-dessus.

Une fois qu'on dispose de la liste des suffixes, l'idée est que chaque recherche dans `T` peut se faire par dichotomie, et donc avec une complexité en $O(\ln(|T|))$, bien meilleure que la méthode naïve.

▶ **Question 7** : écrire une fonction `comparerMotSuffixe(a,b)` qui renvoie :

- 0 si `a` est un préfixe de `b`
- 1 si `b` est inférieur à `a` pour l'ordre lexicographique, et que `a` n'est pas préfixe de `b`

- −1 si `a` est inférieur à `b` pour l'ordre lexicographique, et que `a` n'est pas préfixe de `b`.

⚠ Attention : `'bon'` est inférieur à `'bonbon'` pour l'ordre lexicographique !

▶ **Question 8** : en déduire alors une fonction `dichoSuffixes(M)`, qui utilise la fonction précédente et une recherche dichotomique dans la table des suffixes pour déterminer si le motif `M` apparaît ou non dans le texte `T`.

▶ **Question 9** : l'algorithme de tri que Python utilise lorsqu'on fait appel à `sort` est quasi-linéaire, ce qui signifie que le nombre de comparaisons de chaînes effectuées pour trier la liste est de l'ordre de $n\ln(n)$, où n est la longueur de la chaîne. Toutefois, ici ces comparaisons sont des comparaisons de chaînes, qui peuvent être longues dans le cas de grandes chaînes.
Montrer que la complexité de la création de la liste des suffixes (triée) est en $O(n^2 \ln n)$.

▶ **Question 10** : la création de la liste des suffixes étant coûteuse, elle ne sera intéressante que si on effectue un grand nombre de recherches, chacune de ces recherches ayant une complexité logarithmique.
Le code donné dans le fichier réponse propose de comparer les temps d'exécutions de la recherche naïve et de la recherche par suffixe lors de la recherche de N gènes de longueur 100 bases dans un génome de longueur 50 000.
Déterminer à partir de quelle valeur de N il devient intéressant d'utiliser une table des suffixes. *Restez raisonnables et augmentez N petit à petit ! En tous cas, n'utilisez pas de très grandes valeurs de N pour la méthode naïve.*

En pratique, il existe des moyens de générer une table des suffixes en temps linéaire, et qui rendent cette méthode de recherche bien plus intéressante.

▶ Compter les occurrences

Dans cette partie, on continue d'utiliser la table des suffixes, et on souhaite adapter la recherche dichotomique afin de déterminer combien de fois un motif apparaît dans un texte.
▶ **Question 11** : écrire une fonction `recherchePremierSuffixe(M)` qui retourne la position (dans la liste des suffixes) du premier suffixe commençant par `M`.
On veillera à ce que cette fonction possède toujours une complexité en $O(\ln n)$.
Écrire de même une fonction `rechercheDernierSuffixe(M)` qui retourne la position du dernier suffixe commençant par `M`.

▶ **Question 12** : en déduire une fonction qui compte le nombre d'apparitions de `M` dans le texte `T`.