ANTILLES-GUYANE – 2026 – sujet1

EXERCICE 2 (6 POINTS)

Cet exercice porte sur l'algorithmique et la programmation Python.

La recherche d'une plus longue sous-liste commune à deux listes est un problème connu sous le nom de « Longest Common Subsequence » (LCS).

Les éléments d'une plus longue sous-liste commune sont des éléments communs aux deux listes initiales, où ils apparaissent dans le même ordre, mais pas nécessairement de façon contiguë. Ainsi, les listes `L_1 = [9, 3, 7, 5, 8]` et `L_2 = [9, 7, 8, 3, 7, 3]` ont notamment en commun les sous-listes `[]`, `[9]`, `[3]`, `[9, 7]`, `[9, 3, 7]` ou `[9, 7, 8]`. Les plus longues sous-listes communes à ces deux listes sont `[9, 3, 7]` et `[9, 7, 8]`, de longueur commune égale à 3.

En Python, il est possible d'obtenir la concaténée de deux valeurs de types `list` en utilisant +. Par exemple, `[1, 2] + [3, 4]` s'évalue à `[1, 2, 3, 4]` et `[] + [1]` s'évalue à `[1]`.

PARTIE A

Dans cette partie, on s'attache à déterminer une LCS selon la stratégie de « recherche exhaustive ». La première fonction implémentée construit toutes les sous-listes d'une liste passée en paramètre, en appliquant l'algorithme suivant :

```
Fonction sous_listes(L: list, accumulateur: list[list])
         -> list[list]
Début
| Si L est vide alors
| | renvoyer l'accumulateur (initialement égal à [[]])
| Sinon
| | Extraire e le dernier élément de L
| | Créer L_t une liste temporaire vide
| | Ajouter dans L_t chaque sous-liste présente dans
| | l'accumulateur, après lui avoir ajouté, en tête, e
| | Renvoyer le résultat d'un nouvel appel à la fonction
| | avec pour paramètres
| | - la liste diminuée de cet élément e
| | - l'union des éléments de l'accumulateur et de la
| | liste temporaire
| FinSi
Fin
```

1. Recopier et compléter le tableau suivant, en indiquant la valeur des paramètres lors des appels récursifs, pour un appel initial `sous_listes([9, 3, 7], [[]])`.

Évolution des paramètres lors des appels récursifs		
Appel	**liste**	**accumulateur**
appel initial	[9, 3, 7]	[[]]
1er sous-appel	[9, 3]	[[], [7]]
2nd sous-appel		
dernier sous-appel		

La fonction suivante implémente l'algorithme étudié précédemment.

```
1 def sous_listes(liste: list, accumulateur = [[]]):
2     if len(liste) == 0:
3         return accumulateur
4     element = liste.pop()
5     tmp = []
6     for sous_liste in accumulateur:
7         tmp.append(...)
8     return sous_listes(...)
```

2. Recopier et compléter les lignes 7 et 8 du code de la fonction `sous_listes`.

On implémente la fonction `est_extraite` ci-dessous qui renvoie `True` si une sous-liste peut être extraite d'une liste donnée et `False` sinon.

```
def est_extraite(sous_liste: list, liste: list):
    index_ss_liste, index_liste = 0, 0
    while index_ss_liste < len(sous_liste) and index_liste < len(liste):
        if sous_liste[index_ss_liste] == liste[index_liste]:
            index_ss_liste += 1
        index_liste += 1
    return index_ss_liste == len(sous_liste)
```

On implémente la fonction `lcs_force_brute` ci-dessous qui détermine une plus longue sous-liste commune à deux listes données et qui renvoie cette liste.

```
def lcs_force_brute(liste_1: list, liste_2: list):
    tmp = sous_listes(liste_1)
    resultat, maxi = [], 0
    for element in tmp:
        if est_extraite(element, liste_2) and len(element) > maxi:
            resultat = element
            maxi = len(element)
    return resultat
```

3. Expliquer pourquoi l'exécution de la fonction `lcs_force_brute` risque de prendre beaucoup de temps selon la longueur des listes passées en paramètre. Justifier.

PARTIE B

Soient m et n deux entiers naturels, m' et n' des entiers tels que $0 \leq m' < m$ et $0 \leq n' < n$. On suppose, dans cette partie, disposer d'une fonction `lcs` qui, pour deux listes quelconques `L_1` et `L_2`, de tailles respectives m et n, et des entiers m' et n' passés en paramètres, renvoie une plus longue sous-liste commune à :

- la sous-liste de `L_1` constituée des m' premiers éléments ;
- et la sous-liste de `L_2` constituée des n' premiers éléments.

Ainsi,

- `lcs([9, 3, 7, 5, 8], 0, [9, 7, 8, 3, 7, 3], 0)` est une plus longue sous-liste commune à deux listes vides, soit `[]` ;
- `lcs([9, 3, 7, 5, 8], 3, [9, 7, 8, 3, 7, 3], 2)` est une plus longue sous-liste commune à `[9, 3, 7]` et `[9, 7]`, soit la sous-liste `[9, 7]`.

4. Indiquer ce que renvoie l'appel suivant :

```
lcs([9, 3, 7, 5, 8], 1, [9, 7, 8, 3, 7, 3], 1)
```

On suppose, dans cette question, que `L_1` et `L_2` sont des listes non vides dont les derniers éléments sont identiques : $m \geq 1$, $n \geq 1$, `L_1[m-1]` et `L_2[n-1]` égaux.

Exemple : $m = 5$, $n = 3$, `L_1 = [9, 3, 7, 5, 8]` et `L_2 = [9, 7, 8]`.

5. Écrire la relation de récurrence pour obtenir une plus longue sous-liste commune aux deux listes `L_1` et `L_2` à partir du résultat de `lcs(L_1, m-1, L_2, n-1)`.

A contrario, lorsque les derniers éléments des listes sont différents ($m \geq 1$, $n \geq 1$, `L_1[m-1]` et `L_2[n-1]` distincts), une plus longue sous-suite commune aux deux listes `L_1` et `L_2` s'obtient en choisissant une plus grande liste parmi les résultats de `lcs(L_1, m-1, L_2, n)` et `lcs(L_1, m, L_2, n-1)`.

Par exemple, avec $m = 5$, $n = 4$, `L_1 = [9, 3, 7, 5, 8]` et `L_2 = [9, 7, 8, 3]`, on compare les longueurs d'une LCS de `[9, 3, 7, 5]` et `L_2 = [9, 7, 8, 3]` (soit `[9, 7]`) et d'une LCS de `L_1 = [9, 3, 7, 5, 8]` et `[9, 7, 8]` (soit `[9, 7, 8]`), ce qui permet d'obtenir la LCS `[9, 7, 8]` pour `L_1` et `L_2`.

On implémente donc la fonction suivante qui prend en paramètres deux listes et qui renvoie une plus longue sous-liste commune :

```
def lcs_top_down(liste_x: list, liste_y: list):
    def aux(liste_x, index_x, liste_y, index_y):
        if (index_x < 0) or (index_y < 0):
            return []
        if liste_x[index_x] == liste_y[index_y]:
            return aux(liste_x, index_x-1, liste_y, index_y-1) + [liste_x[index_x]]
        resultat_1 = aux(liste_x, index_x, liste_y, index_y-1)
        resultat_2 = aux(liste_x, index_x-1, liste_y, index_y)
        if len(resultat_1) >= len(resultat_2):
            return resultat_1
        return resultat_2
    return aux(liste_x, len(liste_x)-1, liste_y, len(liste_y)-1)
```

6. Justifier que la fonction `aux` contenue dans la fonction `lcs_top_down` est récursive.

7. Détailler le risque d'un tel choix dans la présente situation, lorsque les listes `L_1` et `L_2` sont très grandes.

8. Proposer une solution pour éviter le problème évoqué à la question 7.