Antilles-Guyane – 2026 – sujet1 - Correction

Exercice 2 (6 points)

Partie A – Recherche exhaustive

1. Évolution des paramètres

Appel	liste	accumulateur
appel initial	[9, 3, 7]	[[]]
1er sous-appel	[9, 3]	[[], [7]]
2nd sous-appel	**[9]**	**[[], [7], [3], [3, 7]]**
dernier sous-appel	**[]**	**[[], [7], [3], [3, 7], [9], [9, 7], [9, 3], [9, 3, 7]]**

👉*Commentaire : à chaque appel, on retire le dernier élément e de la liste et on double l'accumulateur : on garde les anciennes sous-listes et on ajoute les mêmes précédées de e. Avec 3 éléments, on obtient $2^3 = 8$ sous-listes.*

2. Lignes 7 et 8 de sous_listes

```
def sous_listes(liste: list, accumulateur = [[]]):
    if len(liste) == 0:
        return accumulateur
    element = liste.pop()
    tmp = []
    for sous_liste in accumulateur:
        tmp.append([element] + sous_liste)         # ligne 7
    return sous_listes(liste, accumulateur + tmp)  # ligne 8
```

👉*Commentaire : [element] + sous_liste place element en tête, comme le demande l'algorithme. accumulateur + tmp crée une nouvelle liste : la valeur par défaut [[]] n'est donc jamais modifiée.*

👉*Commentaire : liste.pop() vide la liste reçue : après lcs_force_brute(L_1, L_2), la liste L_1 de l'appelant est vide. On pourrait appeler sous_listes(liste_1[:]) pour travailler sur une copie.*

3. Pourquoi lcs_force_brute est lente

Une liste de m éléments possède **2^m sous-listes** (chaque élément est pris ou non). La fonction les construit toutes, puis appelle est_extraite pour chacune, avec un coût proportionnel à la longueur n de liste_2.

Le coût est donc de l'ordre de $\mathbf{2^m \times n}$: il est **exponentiel**. Chaque élément ajouté à liste_1 double le temps de calcul (et la mémoire occupée par les sous-listes).

👉 *Commentaire : pour m = 30, il y a déjà plus d'un milliard de sous-listes ; pour m = 60, plus de 10^{18}.*

Partie B – Approche récursive

4. Valeur de lcs([9, 3, 7, 5, 8], 1, [9, 7, 8, 3, 7, 3], 1)

On cherche une LCS de [9] et de [9] : l'appel renvoie **[9]**.

5. Relation de récurrence (derniers éléments égaux)

lcs(L_1, m, L_2, n) = **lcs(L_1, m-1, L_2, n-1) + [L_1[m-1]]**

Le dernier élément commun termine une plus longue sous-liste commune : on le place à la fin d'une LCS des listes privées de leur dernier élément.

👉 *Commentaire : exemple de l'énoncé : lcs(L_1, 4, L_2, 2) = [9, 7], donc lcs(L_1, 5, L_2, 3) = [9, 7] + [8] = [9, 7, 8].*

6. aux est récursive

La fonction aux **s'appelle elle-même** (lignes 6, 7 et 8), avec des indices plus petits. Le cas de base (ligne 3 : un indice négatif, c'est-à-dire une liste vide) arrête les appels.

7. Risque de ce choix pour de très grandes listes

- **Temps de calcul exponentiel** : lorsque les derniers éléments diffèrent, aux fait **deux appels**, qui recalculent à de nombreuses reprises les mêmes sous-problèmes aux(i, j). Le nombre d'appels peut aller jusqu'à environ 2^{m+n}.
- **Dépassement de la pile d'appels** : la profondeur de récursion peut atteindre m + n. Python limite cette profondeur (1000 par défaut) : au-delà, le programme s'arrête avec une erreur RecursionError.

👉 *Commentaire : en exécutant lcs_top_down sur deux listes sans élément commun, on compte 25 739 appels de aux pour 8 éléments chacune et plus de 80 millions pour 14 éléments.*

8. Solution : la programmation dynamique

On évite les calculs répétés en **mémorisant** les résultats des sous-problèmes. Le mieux ici est une version **ascendante** (bottom-up), **itérative**, qui remplit un tableau t où t[i][j] est une LCS des i premiers éléments de liste_x et des j premiers de liste_y : plus d'appels récursifs, donc plus de risque de dépassement de pile, et un coût en O(m × n) cases.

```python
def lcs_bottom_up(liste_x, liste_y):
    m, n = len(liste_x), len(liste_y)
    t = [[[] for j in range(n + 1)] for i in range(m + 1)]
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if liste_x[i - 1] == liste_y[j - 1]:
                t[i][j] = t[i - 1][j - 1] + [liste_x[i - 1]]
            elif len(t[i][j - 1]) >= len(t[i - 1][j]):
                t[i][j] = t[i][j - 1]
            else:
                t[i][j] = t[i - 1][j]
    return t[m][n]
```

👉*Commentaire : lcs_bottom_up([9, 3, 7, 5, 8], [9, 7, 8, 3, 7, 3]) renvoie [9, 7, 8], comme lcs_top_down. On peut aussi garder la version récursive en ajoutant un dictionnaire memo de clés (index_x, index_y) : le temps devient raisonnable, mais la profondeur de récursion reste un problème pour de très grandes listes.*