TD : PROGRAMMATION DYNAMIQUE
== PLUS LONGUE SOUS-SUITE COMMUNE (LCS) ==

Remarque : les rappels théoriques sont en dernière page de ce sujet.

Le fichier source à utiliser pour ce TD est : « TD3 – LCS.py »

Vous travaillez dans un laboratoire de bio-informatique. Votre mission est de développer un outil de comparaison de séquences ADN. Le principe est le suivant :

- On dispose de deux séquences ADN (chaînes de caractères composées de A, C, G, T) ;
- On cherche la plus longue sous-suite commune (LCS) entre ces deux séquences ;
- Cette LCS représente les parties communes héritées d'un ancêtre commun ;
- On affiche la sous-suite commune reconstruite pour identifier les gènes partagés.

Exemple : Soient deux séquences S1 = "ACGTAC" et S2 = "AGTCAT". La plus longue sous-suite commune est "AGTA" (longueur 4). Cette sous-suite représente les nucléotides communs, dans le même ordre, entre les deux séquences.

L'objectif de ce TD est d'implémenter les algorithmes de programmation dynamique (approches bottom-up et top-down) pour calculer la longueur de la LCS, puis reconstruire la sous-suite commune. Vous utiliserez des dictionnaires Python pour mémoriser les résultats des sous-problèmes.

I) APPROCHE BOTTOM-UP (TABULATION)

Dans cette partie, vous allez implémenter l'approche bottom-up qui remplit une table de tous les sous-problèmes, des plus petits aux plus grands.

On utilisera un dictionnaire `L` pour stocker les valeurs `L[(i, j)]` représentant la longueur de la LCS entre les i premiers caractères de S1 et les j premiers caractères de S2.

Les données sont déjà définies dans le fichier source :

```
# Séquences ADN à comparer
seq1 = "ACGTAC"
seq2 = "AGTCAT"
L = {}  # Table de mémoïsation
```

1. Écrire une fonction `initialiser_cas_de_base(S1, S2, L)` qui initialise et retourne le dictionnaire `L` avec les cas de base.

   Tester :
   ```
   >>> initialiser_cas_de_base(seq1,seq2,L)
   {(0, 0): 0, (1, 0): 0, (2, 0): 0, (3, 0): 0, (4, 0): 0, (5, 0): 0,
   (6, 0): 0, (0, 1): 0, (0, 2): 0, (0, 3): 0, (0, 4): 0, (0, 5): 0,
   (0, 6): 0}
   ```

2. Écrire une fonction `remplir_table(S1, S2, L)` qui remplit entièrement la table `L` en utilisant l'équation de récurrence (voir rappels à la fin du sujet). L'ordre de parcours est : pour i allant de 1 à n, et pour chaque i, j allant de 1 à m.

Vérifier :
```
>>> L = initialiser_cas_de_base(seq1,seq2,L)
>>> L = remplir_table(seq1,seq2,L)
>>> AfficheTable(seq1,seq2,L)
```

Préfixe S2

	j	0	1	2	3	4	5	6
i			A	G	T	C	A	T
0		0	0	0	0	0	0	0
1	A	0	1	1	1	1	1	1
2	C	0	1	1	1	2	2	2
3	G	0	1	2	2	2	2	2
4	T	0	1	2	3	3	3	3
5	A	0	1	2	3	3	4	4
6	C	0	1	2	3	4	4	4

Préfixe S1

3. Écrire une fonction `lcs_bottomup(S1, S2)` qui utilise les fonctions précédentes pour calculer et retourner la table L et la longueur de la LCS entre S1 et S2.

Vérifier :
```
>>> seq1 = "ACGTAC"
>>> seq2 = "AGTCAT"
>>> lcs_bottomup(seq1,seq2)
4
```

4. Écrire une fonction `comparer_sequences(seq_reference, liste_sequences)` qui compare une séquence de référence à une liste de séquences candidates. La fonction retourne la ou les séquences ayant la plus grande LCS avec la référence, ainsi que la longueur maximale.

Vérifier :
```
>>> reference = "ACGTAC"
>>> candidates = ["ACGT", "CGTA", "GTAC", "TACG"]
>>> comparer_sequences(reference,candidates)
(['ACGT', 'CGTA', 'GTAC'], 4)
```

5. Combien de sous-problèmes sont calculés dans l'approche bottom-up pour comparer deux chaînes de longueurs n et m ? Quelle est la complexité temporelle et spatiale de cet algorithme ?

II) APPROCHE TOP-DOWN AVEC MÉMOÏSATION

Dans cette partie, vous allez implémenter l'algorithme récursif avec mémoïsation. L'idée est de partir du problème principal L[(n, m)] et de le décomposer en sous-problèmes, en mémorisant les résultats pour éviter les calculs redondants.

On utilisera un dictionnaire défini dans le programme principal pour la mémoïsation : L = {}

1. Écrire une fonction récursive `rec_lcs(S1, S2)` qui implémente la récurrence rappelée à la fin du sujet. Voici un exemple que vous pouvez suivre si vous le souhaitez :

```
L = {}
def rec_lcs(S1, S2):
    n = ...
    m = ...
    def f_rec(i, j):
        # Utilise la mémoïsation
        if ... in L:
            return ...

        # Cas de base
        if i == 0 or j == 0:
            L[(i, j)] = ...
            return ...

        # Cas du match
        if ..............:
            L[(i, j)] = ............
            return .........

        # Sinon, calcule les deux possibilités
        else:
            V1 = ..............
            V2 = ..............

            # Mémoïse et retourne la valeur optimale
            L[(i, j)] = ..............
            return L[(i, j)]

    longueur = f_rec(n, m)
    return longueur
```

Préfixe S2 / Préfixe S1

	j	0	1	2	3	4	5	6
i			A	G	T	C	A	T
0		0		0	0	0		
1	A		1	1	1		1	
2	C	0	1	1	1	2	2	
3	G	0	1	2	2	2	2	
4	T	0	1	2	3	3		3
5	A		1	2	3		4	4
6	C					4	4	4

Tester :

```
>>> rec_lcs(seq1,seq2)
4
```

Vérifier la table avec : `>>> AfficheTable(seq1,seq2,L)`

2. Comparez les tables obtenues avec les approches bottom-up et top-down (optimisée). Pourquoi l'approche top-down optimisée calcule-t-elle moins de sous-problèmes ? Dans quel cas cette différence serait-elle plus marquée ?

3. Quelle est la complexité temporelle de l'algorithme top-down avec mémoïsation dans le pire cas ? Quelle est la complexité spatiale (dictionnaire + pile d'appels) ?

III) RECONSTRUCTION DE LA SOLUTION

Maintenant que nous savons calculer la longueur de la LCS, nous devons reconstruire la sous-suite commune elle-même. Cette étape permet d'identifier précisément quels nucléotides sont partagés entre les deux séquences ADN.

La reconstruction consiste à « remonter » dans la table L depuis L[(n, m)] jusqu'à L[(0, 0)] pour déterminer, à chaque étape, quel choix a été fait (voir les rappels à la fin du sujet).

Attention avec l'approche top-down utilisée ici : Lors de la reconstruction, on doit comparer L[(i, j)] avec ses voisins L[(i-1, j-1)], L[(i-1, j)] et L[(i, j-1)]. Or, avec notre algorithme top-down, certaines de ces valeurs n'ont pas été calculées ! En effet, quand il y a un match, seul le cas diagonal L[(i-1, j-1)] est exploré.

Pour éviter ce problème, il faut tester le cas du match en priorité (cas diagonal avec égalité des caractères) avant de tester les autres cas. Ainsi, quand une valeur a été obtenue par un match, on la détecte immédiatement sans jamais essayer d'accéder aux clés inexistantes.

1. Écrire une fonction `determiner_choix(S1, S2, L, i, j)` qui retourne le choix effectué pour arriver à L[(i, j)].
   Cette fonction doit retourner un tuple `(choix, new_i, new_j)` où :
   - `choix` est une chaîne décrivant le choix ("GARDE X", "LAISSE X", "LAISSE X") ;
   - `new_i` et `new_j` sont les nouveaux indices après le choix.

```
Tester :    >>> determiner_choix(seq1,seq2,L,6,6)
            ('LAISSE C', 5, 6)
            >>> determiner_choix(seq1,seq2,L,5,5)
            ('GARDE A', 4, 4)
            >>> determiner_choix(seq1,seq2,L,4,3)
            ('GARDE T', 3, 2)
```

2. Écrire une fonction reconstruire_lcs(S1, S2, L) qui retourne la plus longue sous-suite commune sous forme de chaîne de caractères.

```
Tester :    >>> reconstruire_lcs(seq1,seq2,L)
            AGTA
```

3. Quelle est la complexité temporelle de la reconstruction ?

4. Quelle est la complexité finale {Calcul des valeurs optimales + reconstruction} ?

RAPPELS THÉORIQUES

Formulation du problème

Soient deux chaînes de caractères S1 de longueur n et S2 de longueur m. Une sous-suite (ou sous-séquence) d'une chaîne est obtenue en supprimant zéro, un ou plusieurs caractères, sans changer l'ordre des caractères restants.

La plus longue sous-suite commune (LCS) de S1 et S2 est une sous-suite qui apparaît dans S1 et dans S2, de longueur maximale.

Sous-problèmes et notation

On note $L_{i,j}$ la longueur de la LCS entre les i premiers caractères de S1 (le préfixe S1[1..i]) et les j premiers caractères de S2 (le préfixe S2[1..j]).

Relation de récurrence

Pour tout i ∈ {1..n} et j ∈ {1..m} :

$$L_{i,j} = \begin{cases} L_{i-1,j-1} + 1 & si\ S1[i] == S2[j] \quad (cas\ n°1\ -\ match) \\ max \begin{cases} L_{i-1,j} \\ L_{i,j-1} \end{cases} & sinon \qquad\qquad (cas\ n°\ 2\ -\ pas\ de\ match) \end{cases}$$

Cas de base

Les cas de base sont les suivants :

- $L_{0,j} = 0$: la chaîne vide n'a aucun caractère commun avec S2.
- $L_{i,0} = 0$: S1 n'a aucun caractère commun avec la chaîne vide.

Algorithme de reconstruction

Une fois la table des valeurs optimales remplie, on reconstruit la solution en « remontant » depuis $L_{n,m}$ jusqu'à $L_{0,0}$.

Principe : À chaque position (i, j), on détermine quel choix a permis d'obtenir $L_{i,j}$ en comparant avec les valeurs voisines :

- Si $L_{i,j} == L_{i-1,j-1} + 1$ ET S1[i] == S2[j] → Match (on ajoute le caractère à la LCS)
- Si $L_{i,j} == L_{i-1,j}$ → On ignore le caractère de S1
- Si $L_{i,j} == L_{i,j-1}$ → On ignore le caractère de S2