TD : PROGRAMMATION DYNAMIQUE
== DISTANCE DE LEVENSHTEIN ==

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

Le fichier source à utiliser pour ce TD est : « TD4 – Levenshtein.py »

Vous développez un mini-correcteur orthographique. Le principe est le suivant :
- L'utilisateur tape un mot (potentiellement mal orthographié) ;
- On compare ce mot à chaque mot d'un dictionnaire de mots corrects ;
- On calcule la distance d'édition (distance de Levenshtein) entre le mot tapé et chaque mot du dictionnaire ;
- On suggère le mot du dictionnaire le plus proche (distance minimale) ;
- On affiche les opérations de correction pour transformer le mot erroné en mot correct.

Exemple : L'utilisateur tape "ALGORYTME". Le correcteur compare ce mot au dictionnaire et trouve que "ALGORITHME" est le mot le plus proche (distance = 2). Il suggère alors la correction avec les opérations : substituer 'Y' par 'I' et insérer 'H'.

L'objectif de ce TD est d'implémenter les algorithmes de programmation dynamique (approches bottom-up et top-down) pour calculer la distance d'édition, trouver le mot le plus proche, puis reconstruire la suite d'opérations de correction. 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 D pour stocker les valeurs D[(i, j)] représentant la distance d'édition entre les i premiers caractères de S1 et les j premiers caractères de S2.

Les données sont déjà définies dans le fichier source :

```
mot_utilisateur = "camiont"
dictionnaire = ["camion","camions","canon","cation","canton","camionne"]
D = {}    # Table de mémoïsation
```

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

   Tester :
   ```
   >>> initialiser_cas_de_base("tu","toi",D)
   {(0, 0): 0, (1, 0): 1, (2, 0): 2, (0, 1): 1, (0, 2): 2,
   (0, 3): 3}
   ```

2. Écrire une fonction `remplir_table(S1, S2, D)` qui remplit entièrement la table D 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 :  S1 = "ALGORYTME"
            S2 = "ALGORITHME"
            D = {}
            D = initialiser_cas_de_base(S1,S2,D)
            D = remplir_table(S1,S2,D)
            AfficheTable(S1,S2,D)
```

Préfixe S2

	j	0	1	2	3	4	5	6	7	8	9	10
i			A	L	G	O	R	I	T	H	M	E
0		0	1	2	3	4	5	6	7	8	9	10
1	A	1	0	1	2	3	4	5	6	7	8	9
2	L	2	1	0	1	2	3	4	5	6	7	8
3	G	3	2	1	0	1	2	3	4	5	6	7
4	O	4	3	2	1	0	1	2	3	4	5	6
5	R	5	4	3	2	1	0	1	2	3	4	5
6	Y	6	5	4	3	2	1	1	2	3	4	5
7	T	7	6	5	4	3	2	2	1	2	3	4
8	M	8	7	6	5	4	3	3	2	2	2	3
9	E	9	8	7	6	5	4	4	3	3	3	2

Préfixe S1

3. Écrire une fonction `distance_levenshtein_bottomup(S1, S2)` qui utilise les fonctions précédentes pour calculer et retourner la table D et la distance d'édition entre S1 et S2.

```
Tester :    >>> S1 = "ALGORYTME"
            >>> S2 = "ALGORITHME"
            >>> D, distance = distance_levenshtein_bottomup(S1,S2)
            >>> print(distance)
            2
```

4. Écrire une fonction `trouver_mot_proche(mot_utilisateur, dictionnaire)` qui parcourt tous les mots du dictionnaire, calcule la distance avec le mot de l'utilisateur, et retourne le ou les mots les plus proches ainsi que la distance minimale.

```
Tester :    >>> trouver_mot_proche(mot_utilisateur,dictionnaire)
            (['camion', 'camions'], 1)
```

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 D[(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 : D = {}

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

```
D = {}
def rec_levenshtein(S1,S2,D):
    n = ...
    m = ...

    def f_rec(i,j):
        # Utilise la mémoisation
        if ... in D:
            return ...

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

        # Test si match
        if ..............:
            D[(i,j)] = ............
            return .........

        # Sinon, calcule les trois autres possibilités
        else:
            V1 = ..............
            V2 = ..............
            V3 = ..............

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

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

```
Tester :    S1 = "ALGORYTME"
            S2 = "ALGORITHME"
            >>> rec_levenshtein(S1,S2,D)
            2
```

Vérifier la table : `>>> AfficheTable(S1,S2,D)`

Préfixe S2

	j	0	1	2	3	4	5	6	7	8	9	10
i			A	L	G	O	R	I	T	H	M	E
0		0	1	2	3	4	5	6	7	8		
1	A	1	0	1	2	3	4	5	6	7		
2	L	2	1	0	1	2	3	4	5	6		
3	G	3	2	1	0	1	2	3	4	5		
4	O	4	3	2	1	0	1	2	3	4		
5	R	5	4	3	2	1	0	1	2	3		
6	Y	6	5	4	3	2	1	1	2	3		
7	T								1	2		
8	M										2	
9	E											2

Préfixe S1

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 trouver le mot le plus proche et calculer la distance d'édition, nous devons expliquer à l'utilisateur comment corriger son mot. Cette étape s'appelle la reconstruction de la solution.

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

Attention avec l'approche top-down optimisée : Lors de la reconstruction, on doit comparer D[(i, j)] avec ses voisins D[(i-1, j-1)], D[(i-1, j)] et D[(i, j-1)]. Or, avec l'algorithme top-down optimisé, certaines de ces valeurs n'ont pas été calculées ! En effet, quand il y a un match, seul le cas diagonal D[(i-1, j-1)] est exploré, les cas D[(i-1, j)] et D[(i, j-1)] ne sont jamais calculés et n'existent donc pas dans le dictionnaire.

Pour éviter ce problème, il faut tester les cas diagonaux en priorité (match puis substitution) avant de tester les cas de suppression et d'insertion. 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_operation(S1, S2, D, i, j)` qui retourne l'opération effectuée pour arriver à D[(i, j)]. Cette fonction doit retourner un tuple `(operation, new_i, new_j)` où :
   - `operation` est une chaîne décrivant l'opération ("SUPPR X", "INSERT X", "GARDER X", "SUBST X <- Y")
   - `new_i` et `new_j` sont les nouveaux indices après l'opération

   Vérifier :
   ```
   >>> S1 = "ALGORYTME"
   >>> S2 = "ALGORITHME"
   >>> rec_levenshtein(S1,S2)
   2
   >>> determiner_operation(S1,S2,D,9,10)
   ('GARDER E', 8, 9)
   >>> determiner_operation(S1,S2,D,7,8)
   ('INSERT H', 7, 7)
   >>> determiner_operation(S1,S2,D,6,6)
   ('SUBST Y<-I', 5, 5)
   ```

2. Écrire une fonction `reconstruire_operations(S1, S2, D)` qui retourne la liste des opérations à effectuer (dans l'ordre) pour transformer S1 en S2.

   Vérifier :
   ```
   >>> reconstruire_operations(S1,S2,D)
   ['GARDER A', 'GARDER L', 'GARDER G', 'GARDER O', 'GARDER
   R', 'SUBST Y<-I', 'GARDER T', 'INSERT H', 'GARDER M',
   'GARDER E']
   ```

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. La distance d'édition (ou distance de Levenshtein) est le nombre minimal d'opérations élémentaires pour transformer S1 en S2.

Les trois opérations autorisées, chacune de coût 1, sont l'insertion d'un caractère dans S1, la suppression d'un caractère de S1 et la substitution d'un caractère de S1 par un autre. Garder un caractère (match) a un coût nul.

Sous-problèmes et notation

On note $D_{i,j}$ la distance d'édition minimale 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} :

$$D_{i,j} = min \begin{cases} D_{i-1,j} + 1 & (cas\ n°1 : suppression) \\ D_{i,j-1} + 1 & (cas\ n°3 : insertion) \\ D_{i-1,j-1} + \begin{cases} 0, & si\ S_1[i] == S_2[j] \\ 1, & sinon \end{cases} & (cas\ n°2 : substitution\ /\ match) \end{cases}$$

Cas de base

Les cas de base sont les suivants :

- $D_{0,j} = j$: transformer la chaîne vide en j caractères demande j insertions.
- $D_{i,0} = i$: transformer i caractères en chaîne vide demande i suppressions.

Algorithme de reconstruction

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

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

- Si $D_{i,j} == D_{i-1,j-1}$ ET S1[i] == S2[j] → Match
- Si $D_{i,j} == D_{i-1,j-1} + 1$ → Substitution
- Si $D_{i,j} == D_{i-1,j} + 1$ → Suppression
- Si $D_{i,j} == D_{i,j-1} + 1$ → Insertion