Exercice 3 : Alignement de séquences

L'**alignement de séquences** est un problème algorithmique classique de bio-informatique visant typiquement à comparer deux séquences d'ADN dans l'idée que si deux séquences d'ADN représentant des gènes sont similaires, alors la fonction du gène est probablement la même.

Concrètement, un alignement de séquences est une manière de représenter deux ou plusieurs séquences (ici des chaînes de caractères) les unes sous les autres, de manière à en faire ressortir les régions similaires et différentes. Chaque alignement possède un coût.

Étant données deux chaînes de caractères, comment les aligner de manière à faire correspondre le plus de régions similaires ?

On peut représenter deux séquences comme deux chaînes de caractères T_1 et T_2 et l'objectif est d'étudier leur degré de similarité (ou dissimilarité). Il y a deux approches possibles :

- soit on étudie leur dissimilarité : on cherche donc à minimiser la distance $d(T_1, T_2)$ entre les deux chaînes ;
- soit on étudie leur similarité : on cherche donc à maximiser un score de ressemblance $s(T_1, T_2)$ entre les deux chaînes.

Dans cet exercice, on choisira la première option : **minimiser la distance entre deux chaînes**.

1 Distance d'édition et alignement

Distance d'édition

Pour mesurer la distance entre deux chaînes, on choisit d'utiliser la **distance d'édition**, aussi appelée distance de Levenshtein.

L'objectif est de passer d'une chaîne à l'autre (de T_1 à T_2) au moyen de 3 opérations :

- insertion (d'un caractère c dans T_1)
- suppression (d'un caractère c dans T_1)
- substitution (d'un caractère c par un autre caractère c' dans T_1)

Chacune de ces opérations a un coût, et on **supposera dans la suite que les coûts des trois opérations sont tous égaux à 1 !!**.

La distance d'édition, notée dE, minimise la somme de ces coûts pour passer d'une chaîne à l'autre. Autrement dit, la distance d'édition entre T_1 et T_2 est égale au nombre minimal de caractères qu'il faut insérer, supprimer, ou substituer pour passer de T_1 à T_2.

Exemple : T_1 = SUCCES et T_2 = ECHECS. On peut considérer l'alignement suivant :

```
S U C C E - S
- E C H E C S
```

Explication de cette notation :

- les tirets (-) sur le premier mot correspondent à une insertion
- les tirets (-) sur le deuxième mot correspondent à une suppression
- si deux caractères sont alignés :
  - soit il s'agit d'une substitution s'ils sont différents,
  - soit il s'agit d'une correspondance s'ils sont égaux (le coût est nul dans ce cas).

Autrement dit, avec cet alignement, pour passer de SUCCES à ECHECS, on a effectué les opérations suivantes :

- suppression de S : -UCCES (coût = 1)
- substitution de U par E : ECCES (coût = 1)
- correspondance de C : ECCES
- substitution de C par H : ECHES (coût = 1)
- correspondance de E : ECHES
- insertion de C : ECHECS (coût = 1)
- correspondance de S : ECHECS

On obtient un coût total égal à 4 (en fait il s'agit d'un alignement optimal, donc dE(SUCCES, ECHECS) = 4, comme on le verra plus tard).

Pour bien comprendre le principe, avant de le programmer, on commence par un exemple à dérouler à la main !

Question 1 : T_1 = SUCCES et T_2 = ECHECS. On considère l'alignement suivant

```
S - U C C E - S
- E - C H E C S
```

1. Écrivez les opérations pour passer de SUCCES à ECHECS avec cet alignement.
2. Quel est le coût de cet alignement ? Cet alignement est-il meilleur que le précédent ?

Moralité

Plusieurs alignements sont possibles et parmi ces alignements, certains ont un coût minimal : il s'agit d'alignements optimaux dont le coût est égal à la distance d'édition entre les deux chaînes. Ainsi, trouver un alignement optimal est équivalent à déterminer la distance d'édition.

2 Relation de récurrence

Le calcul de la distance d'édition entre deux chaînes peut se faire à partir de celles de chaînes plus petites. Cela permet de dégager une relation de récurrence qui sera exploitée par les algorithmes calculant la distance d'édition.

Notations : Dans la suite, on notera:

- n_1 la longueur de T_1 ; n_2 la longueur de T_2
- $T_1[i]$ le i-ème caractère de T_1 ; $T_2[j]$ le j-ème caractère de T_2 (avec $1 \leq i \leq n_1$ et $1 \leq j \leq n_2$)
- $T_1[1..i]$ les i premiers caractères de T_1 ; $T_2[1..j]$ les j premiers caractères de la chaîne T_2, (avec $1 \leq i \leq n_1$ et $1 \leq j \leq n_2$)

Comment peut-on calculer $dE(T_1[1..i], T_2[1..j])$ si on connaît $dE()$ pour des "bouts de textes" plus courts ?

Supposons avoir aligné $T_1[1..i]$ et $T_2[1..j]$ (les i premiers caractères de T_1 et les j premiers caractères de T_2) et cherchons le coût de cet alignement. On regarde l'alignement le plus à droite. Quatre cas se présentent :

- Suppression : $T_1[i]$ s'aligne sur « - ». Cela demande à avoir aligné $T_1[1..i-1]$ et $T_2[1..j]$ et supprimé $T_1[i]$. On a donc dans ce cas :

$$\text{coût}(T_1[1..i], T_2[1..j]) = \text{coût}(T_1[1..i-1], T_2[1..j]) + \underbrace{1}_{\text{coût suppression}}$$

- Insertion : « - » s'aligne sur $T_2[j]$. Cela demande à avoir aligné $T_1[1..i]$ et $T_2[1..j-1]$ et inséré $T_2[j]$. On a donc dans ce cas :

$$\text{coût}(T_1[1..i], T_2[1..j]) = \text{coût}(T_1[1..i], T_2[1..j-1]) + \underbrace{1}_{\text{coût insertion}}$$

- Substitution : $T_1[i]$ s'aligne sur $T_2[j]$ avec $T_1[i] \neq T_2[j]$. Cela demande d'avoir aligné $T_1[1..i-1]$ et $T_2[1..j-1]$ et substitué $T_1[i]$ par $T_2[j]$. On a donc dans ce cas :

$$\text{coût}(T_1[1..i], T_2[1..j]) = \text{coût}(T_1[1..i-1], T_2[1..j-1]) + \underbrace{1}_{\text{coût substitution}}$$

- Correspondance : $T_1[i]$ s'aligne sur $T_2[j]$ avec $T_1[i] = T_2[j]$. Cela demande d'avoir aligné $T_1[1..i-1]$ et $T_2[1..j-1]$ et c'est tout (on substitue le caractère par lui-même). On a donc dans ce cas :

$$\text{coût}(T_1[1..i], T_2[1..j]) = \text{coût}(T_1[1..i-1], T_2[1..j-1]) + \underbrace{0}_{\text{sans surcoût}}$$

Calcul de la distance d'édition

Comme la distance d'édition cherche à minimiser le coût d'un alignement, il faut choisir la valeur minimale de coût à chaque fois (parmi les 4 possibilités). On en déduit la relation de récurrence suivante (on a regroupé les deux derniers cas ensemble en introduisant un coût de substitution qui vaut 1 ou 0 selon les cas) :

$$dE(T_1[1..i], T_2[1..j]) = min \begin{cases} dE(T_1[1..i-1], T_2[1..j]) + 1 \\ dE(T_1[1..i], T_2[1..j-1]) + 1 \\ dE(T_1[1..i-1], T_2[1..j-1]) + \text{sub}(T_1[i], T_2[j]) \end{cases}$$

où $\text{sub}(a,b) = \begin{cases} 0 \text{ si } a = b \text{ (correspondance)} \\ 1 \text{ si } a \neq b \text{ (substitution)} \end{cases}$

3 Version récursive naïve

On peut utiliser cette relation de récurrence pour écrire l'algorithme récursif naïf correspondant. Voici une version avec des slices qui est plus rapide à écrire (mais dont le coût en espace est plus important).

```
# Version récursive naïve
def dE(t1, t2):
    if len(t1) == 0:
        return len(t2)
    elif len(t2) == 0:
        return len(t1)
    else:
        if t1[-1] == t2[-1]:
            sub = 0
        else:
            sub = 1
        return min(dE(t1[0:-1], t2) + 1,
                   dE(t1, t2[0:-1]) + 1,
                   dE(t1[0:-1], t2[0:-1]) + sub)
```

On peut constater que cela marche pour des mots assez courts

```
import time

t_depart = time.time()
distance = dE('SUCCES', 'ECHECS')
t_fin = time.time()
print("distance d'édition =", distance)
print("temps total (en s.) : ", t_fin - t_depart)
```

Que cela prend un peu plus de temps si on augmente un peu la longueur des mots.

```
import time

t_depart = time.time()
distance = dE('DESTRUCTION', 'CREATION')
t_fin = time.time()
print("distance d'édition =", distance)
print("temps total (en s.) : ", t_fin - t_depart)
```

Et que cela ne termine pas en un temps raisonnable (voire pas du tout) si les mots sont encore un peu plus longs.

Attention ! Enregistrez votre travail avant d'exécuter le code suivant !

```
import time

t_depart = time.time()
distance = dE('SOUSTRACTION', 'MULTIPLICATION')
t_fin = time.time()
print("distance d'édition =", distance)
print("temps total (en s.) : ", t_fin - t_depart)
```

Question 2 : Dessinez les 3 premiers niveaux de l'arbre d'appels récursifs si on lance `dE('SUCCES', 'ECHECS')`. On pourra juste noter les deux chaînes en arguments pour gagner un peu de temps.
(Vous devez constater qu'il y a des appels redondants, et qu'ils deviennent vite beaucoup trop nombreux dès que la longueur des chaînes augmente un peu. L'efficacité est catastrophique !)

4 Programmation dynamique

Vous allez maintenant utiliser la programmation dynamique pour améliorer l'algorithme précédent. On ne traitera que la version itérative ascendante (la version récursive avec mémoïsation est laissée en exercice).

Un tableau pour mémoriser

Comme pour les exemples précédents, l'idée est d'utiliser un tableau `d` pour stocker les réponses déjà connues pour ne pas les calculer plusieurs fois.

Dans le cas de l'alignement de séquence :

- on va utiliser un tableau `d` à deux dimensions de taille $(n_1 + 1) \times (n_2 + 1)$.
- Pour tous $1 \leq i \leq n_1$ et pour tous $1 \leq j \leq n_2$, on a : `d[i][j]` = $dE(T_1[1..i], T_2[1..j])$.
  Autrement dit, l'élément en position (i, j) est la distance d'édition entre les i premiers caractères de T_1 et les j premiers caractères T_2.
- La solution $dE(T_1, T_2)$ se trouve dans la case en bas à droite `d[n1][n2]`.

		0	1	2	3	4	5	6
		ε	E	C	H	E	C	S
0	ε							
1	S							
2	U							
3	C							
4	C							
5	E							
6	S							← $dE(T_1, T_2)$ est ici !

Remarques :

- On ajouté une première ligne et une première colonne qui permettront de simplifier le début du remplissage du tableau. Cette première ligne contient ε qui désigne un mot vide.
- Dans cet exemple, la case en position (2,4) c'est-à-dire `d[2][4]` contient la distance d'édition entre les deux premières caractères de `'SUCCES'` et les quatre premiers caractères de `'ECHECS'`, donc elle contient `dE('SU', 'ECHE')`.

Remplissage du tableau

- On peut remplir facilement la première ligne et la première colonne car on cherche à chaque fois la distance avec un mot vide : il faut insérer ou supprimer le nombre de lettres du second mot. On obtient alors :

		0	1	2	3	4	5	6
		ε	E	C	H	E	C	S
0	ε	0	1	2	3	4	5	6
1	S	1						
2	U	2						
3	C	3						
4	C	4						
5	E	5						
6	S	6						

- Il suffit de remplir ensuite le tableau ligne par ligne et de haut en bas en utilisant la relation de récurrence.

Question 3 : Exprimez `d[i][j]` en fonction de `d[i-1][j]`, `d[i][j-1]` et `d[i-1][j-1]` (avec `i` entre 1 et n_1, et `j` entre 1 et n_2) puis identifiez les 4 cases en question dans le tableau `d`.

Question 4 : D'après la réponse à la question précédente, pour compléter une case du tableau on utilise soit celle à sa gauche, soit celle au-dessus, soit celle au-dessus à gauche en diagonale. Selon le cas, indiquez s'il s'agit d'une substitution, d'une suppression, d'une insertion ou d'une correspondance.

Question 5 : Vous devriez avoir tous les éléments pour compléter le tableau `d`. Faites-le et déduisez-en la distance d'édition entre les mots `SUCCES` et `ECHECS`.

Question 6 : Utilisez de même un tableau pour déterminer la distance d'édition entre les mots `NICHE` et `CHIENS`.

		0	1	2	3	4	5	6
		ε	C	H	I	E	N	S
0	ε	0	1	2	3	4	5	6
1	N	1						
2	I	2						
3	C	3						
4	H	4						
5	E	5						

Implémentation

Question 7 : Complétez la fonction `dE(t1, t2)` qui renvoie la distance d'édition entre les chaînes `t1` et `t2` ainsi que le tableau d (cela permettra de vérifier !). On pourra (étape 1) créer le tableau d de départ avec des zéros puis initialiser la première ligne et la première colonne, avant de construire (étape 2) le reste du tableau.

```
def dE(t1, t2):
    # ÉTAPE 1 : Création et initialisation du tableau
    # création du tableau D
    n1 = len(t1)
    n2 = len(t2)
    D = [[0] * (...) for i in range(...)]

    # initialisation première colonne et première ligne
    for i in range(n1 + 1):
        D[i][0] = ...
    for j in range(n2 + 1):
        D[0][j] = ...

    # ÉTAPE 2 : Remplissage du reste du tableau
    for i in range(1, ...):
        for j in range(1, ...):
            if t1[i-1] == ...:  # correspondance
                cout_sub = ...
            else:  # substitution
                cout_sub = 1
            D[i][j] = min(..., ..., ...)

    # ÉTAPE 3 : Le résultat est dans la case en bas à droite
    return D[...][...], D
```

Question 8 : Testez la fonction, en particulier pour vérifier vos réponses aux questions 5 et 6 (distances entre `SUCCES` et `ECHECS`, puis entre `NICHE` et `CHIENS`. Vous vérifierez également que l'on obtient des réponses rapides mêmes pour des mots plus longs.

5 Construction d'un alignement optimal

Notre algorithme nous donne la distance minimale entre deux chaînes de caractères mais ne donne pas un alignement optimal. Peut-on l'adapter pour construire un alignement optimal ?

La réponse est oui ! Pour trouver un alignement optimal entre T_1 et T_2 il suffit de remonter depuis `d[n1][n2]` vers `d[0][0]` (de la case en bas à droite à celle en haut à gauche) en considérant à chaque fois le "meilleur" des 3 voisins, c'est-à-dire celui par lequel on a pu arriver ! (il peut parfois y en avoir plusieurs). Il faut se rappeler (cf. question 4) que :

- Si on monte, il s'agit d'une suppression
- Si on va à gauche, il s'agit d'une insertion
- Si on va en diagonale, il s'agit d'une substitution ou d'une correspondance

Par exemple, le tableau `d` correspondant à l'alignement entre `SUCCES` et `ECHECS` est :

		0	1	2	3	4	5	6
	ε	ε	E	C	H	E	C	S
0	ε	0	1	2	3	4	5	6
1	S	1	1	2	3	4	5	5
2	U	2	2	2	3	4	5	6
3	C	3	3	2	3	4	4	5
4	C	4	4	3	3	4	4	5
5	E	5	4	4	4	3	4	5
6	S	6	5	5	5	4	4	4

Suppression

Insertion

Substitution/Correspondance

← $\mathrm{dE}(T_1, T_2)$ est ici !

On peut alors construire un alignement optimal (en partant de la fin ou du début) :

```
S U C C E - S
- E C H E C S
```

Question 9 : Déterminez un autre alignement optimal de ces deux chaînes.

Question 10 : Utilisez le tableau de la question 6 pour trouver un alignement optimal entre `NICHE` et `CHIENS`. Y en a-t-il d'autres ?

Question 11 (Bonus / DIFFICILE) : Écrivez une fonction `alignement_optimal(t1, t2)` qui renvoie un alignement optimal entre les deux chaînes `t1` et `t2`.

Aide : Contrairement aux deux questions précédentes, plutôt que de "remonter" à partir de la dernière case, il est plus simple de construire l'alignement au fur et à mesure du calcul des différentes valeurs du tableau. Ainsi, on pourra s'inspirer de la fonction `dE` de la question 7, et utiliser un second tableau `sol` qui sera complété au fur et à mesure de sorte que `sol[i][j]` contienne un alignement optimal de `t1[0:i]` et de `t2[0:j]`. La réponse est dans la dernière case du tableau.

Question 12 (Bonus / PLUS FACILE) : Écrivez une fonction permettant de calculer la distance d'édition entre deux chaînes en utilisant une version récursive avec mémoïsation (approche récursive descendante).