

Rappels sur le slicing et les matrices

Le slicing (tranchage en français) permet d'extraire les éléments d'une liste (ou d'une chaîne de caractères) de longueur n qui sont entre deux tranches. Plus précisément, si $0 \leq i \leq j \leq n$, `Liste[i:j]` est la liste des éléments de `Liste` dont les indices sont entre i **inclus** et j **exclus** soit la liste

```
[Liste[i],Liste[i+1],...,Liste[j-1]]
```

de même, `chaîne[i:j]` est la chaîne de caractères extraite de `chaîne` dont l'indice est entre i inclus et j exclus.

1. Essayez et comprenez les exemples suivants :

```
Liste = [3,8,9,10,12]
Slicing1 = Liste[3:4]
Slicing2 = Liste[0:4]
Slicing3 = Liste[1:3]
Slicing4 = Liste[0:3]
chaîne = "mathematiques"
Slicing5 = chaîne[2:5]
```

2. Quelle est la longueur de la chaîne `chaîne[i:j]` ?
3. Que valent `chaîne[0:0]` et `chaîne[0:n]` ?

Une matrice peut être modélisée de deux façons en Python : avec une liste de listes (chaque élément de la liste est une liste représentant une ligne), ainsi, `[[1,2,3],[4,5,6]]` représente $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ ou bien avec des array de numpy.



Attention à la syntaxe

Le coefficient à la ligne i et la colonne j est `M[i][j]` si M est une liste de listes, mais c'est `M[i,j]` si M est un tableau numpy.

Distance de Hamming

Exercice 1. La **distance de Hamming** entre deux chaînes de même longueur est le nombre de caractères différents entre ces deux chaînes. Par

exemple, si `chaîne1 = "AUGCCCUAG"` et `chaîne2 = "AUGUCCUAA"` alors leur distance de Hamming vaut 2. Coder la fonction `Hamming` qui, à deux chaînes de caractères de mêmes longueur, renvoie leur distance de Hamming.

Distance d'édition de Levenshtein

Exercice 2. Dans le prénom `Julie`, si on rajoute quatre lettres et qu'on en enlève une, on tombe sur `ultimate`¹. Dans cet exercice, nous allons considérer qu'un mot peut subir trois types d'opérations élémentaires :

- Un ajout d'une lettre
- Une suppression d'une lettre
- Une modification d'une lettre

Étant donnés deux mots, on peut toujours à partir du premier mot, arriver au second par une succession de modifications, par exemple :

`Julie` → `ulie` → `ultie` → `ultime` → `ultimae` → `ultimate`

Ici, on a fait cinq modifications, on peut se convaincre que c'est le minimum possible. On appelle **distance de Levenshtein** entre `mot1` et `mot2` le nombre minimum d'opérations qu'il faut faire pour passer de `mot1` à `mot2`. Le but est de calculer cette distance sachant qu'on a aucune formule pour l'instant. On pose `n=len(mot1)` et `p=len(mot2)`. Pour $i \in \llbracket 0;n \rrbracket$ et $j \in \llbracket 0;p \rrbracket$, on note `M[i][j]` la distance de Levenshtein entre `mot1[0:i]` et `mot2[0:j]`.

1. Que valent `M[0][j]` et `M[i][0]` ?
2. Que représente `M[n][p]` ?

Soient i dans $\llbracket 1;n \rrbracket$ et j dans $\llbracket 1;p \rrbracket$.

- Si `mot1[i-1]==mot2[j-1]`, alors, il ne sert à rien de toucher à cette lettre, il suffit de transformer `mot1[0:i-1]` en `mot2[0:j-1]` de façon optimale soit

$$M[i][j] = M[i-1][j-1]$$

(par exemple si `mot1[0:i]="Julie"` et `mot2[0:j-1]="Ultimate"` Alors, pour transformer `Julie` en `Ultimate` de façon optimale, il suffit de transformer `Julie` en `Ultimat`, de façon optimale)

- Si `mot1[i-1]!=mot2[j-1]`, (par exemple si `mot1[0:i]="mur"` et `mot2[0:j]="maison"`), alors pour aller de `mot1[0:i]` à `mot2[0:j]`,

1. Coïncidence ? Je ne crois pas.

il faut bien faire quelque chose par rapport à cette dernière lettre, il y a donc trois possibilités (ajout, suppression, modification) :

- Ajouter la dernière lettre de `mot2[0:j]`. Dans ce cas, il faut aller de `mot1[0:i]` à `mot2[0:j-1]` puis rajouter la dernière lettre donc

$$M[i][j] = M[i][j-1] + 1$$

(dans l'exemple, cela veut donc dire qu'on va de `mur` à `maiso` puis on rajoute le `n`).

- Supprimer la dernière lettre de `mot1[0:i]`. Dans ce cas, aller de `mot1[0:i]` à `mot1[0:i-1]` puis de `mot1[0:i-1]` à `mot2[0:j]` donc

$$M[i][j] = 1 + M[i-1][j]$$

(dans l'exemple, cela veut dire que l'on va de `mur` à `mu` puis de `mu` à `maison`).

- Modifier la dernière lettre de `mot1[0:i]` en la dernière lettre de `mot2[0:j]`. Dans ce cas, on transforme `mot1[0:i-1]` en `mot2[0:j-1]` puis on modifie la dernière lettre donc

$$M[i][j] = M[i-1][j-1] + 1$$

(dans l'exemple, on transforme `mu` en `maiso`, par ce chemin, on transforme `mur` en `maisor` puis `maisor` en `maison`).

Comme on ignore si dans le chemin le plus court, il faut faire une délétion, insertion ou une modification pour le dernier caractère, et qu'on veut le moins de modifications possibles, on prend le minimum de ces trois possibilités :

$$M[i][j] = \min(M[i-1][j] + 1, M[i][j-1] + 1, M[i-1][j-1] + 1)$$

Ainsi, si $i \geq 1$ et $j \geq 1$, on a exprimé $M[i][j]$ par une formule de récurrence :

$$M[i][j] = \begin{cases} M[i-1][j-1] & \text{si } \text{mot1}[i-1] = \text{mot2}[j-1] \\ \min(M[i-1][j] + 1, M[i][j-1] + 1, M[i-1][j-1] + 1) & \text{sinon} \end{cases}$$

Programmons une fonction `distance(mot1,mot2)` en plusieurs étapes :

- Créer une matrice `M` destinée à contenir tous les `M[i][j]`.
- Remplir les valeurs de `M[i][0]` et `M[0][j]`.
- Remplir la valeur de `M[i][j]` pour $i \geq 1$ et $j \geq 1$
- Renvoyer la distance entre `mot1` et `mot2`.

3. Créer une fonction `MatriceNulle(n,p)` qui renvoie la matrice nulle à $n + 1$ lignes et $p + 1$ colonnes.
4. Créer une fonction `InitialisationMatrice(n,p)` qui renvoie une matrice `M` à $n + 1$ lignes et $p + 1$ colonnes telles que pour tout $i \in \llbracket 0; n \rrbracket$, $M[i][0] = i$ et pour tout $j \in \llbracket 0; p \rrbracket$, $M[0][j] = j$ et telle que tous les autres coefficients soient nuls.
5. Écrire une fonction `MinimumListe(L)` qui renvoie la plus petite valeur d'une liste `L` non vide de nombres.
6. Écrire la fonction `distance(mot1,mot2)` : initialiser la matrice grâce à la fonction ad-hoc, puis à l'aide d'une double boucle `for`, rentrer la valeur de $M[i][j]$ grâce à la formule de récurrence (ne pas oublier de distinguer les cas).
7. Tester avec `distance("Julie","ultimate")`, puis avec les mots `informatique` et `affirmative`.
8. Récupérer la fonction faite au TP précédent qui génère des séquences aléatoire d'ADN, estimer la distance moyenne de Levenshtein entre deux séquences de 30 nucléotides faites au hasard.
9. Récupérer la liste des mots français du TP7, puis écrire une fonction `ListeMotsPlusProches(chaine)` qui, étant donné une chaîne de caractères, notée `chaine`, renvoie la liste des mots les plus proches (pour la distance de Levenshtein) de `chaine`.
10. Votre professeur d' "imffaurmathique" étant très mauvais en orthographe, trouver la liste des mots français les plus proches de "imffaurmathique". Si vous êtes arrivés là, félicitations, vous venez de programmer un correcteur orthographique !
11. Réécrivez fonction `distance` de façon récursive, que constatez-vous ?
12. On a un long temps de calcul car on a plusieurs fois besoin des mêmes calculs et que Python les refait. Pour y remédier, modifions la récursivité de la façon suivante : stockons les distances déjà calculées : au moment où on a fini de calculer la distance d entre `mot1` et `mot2`, on la rentre dans un dictionnaire (initialement vide) : `Dico[(mot1,mot2)]=d` puis, avant de faire un calcul par récursivité, vérifier si le couple en question n'est pas, par hasard, une clé du dictionnaire, si oui, renvoyer la valeur associée au lieu de refaire le calcul.