Banque « Agro-Véto »
TB - 0726

ALGORITHMIQUE ET INFORMATIQUE

Durée : 45 minutes

L'usage d'abaques, de tables, de calculatrice et de tout instrument électronique susceptible de permettre au candidat d'accéder à des données et de les traiter par les moyens autres que ceux fournis dans le sujet est interdit.
Chaque candidat est responsable de la vérification de son sujet d'épreuve : pagination et impression de chaque page. Ce contrôle doit être fait en début d'épreuve. En cas de doute, le candidat doit alerter au plus tôt le surveillant qui vérifiera et, éventuellement, remplacera le sujet.

Ce sujet comporte 7 pages numérotées de 1 à 7.

Si, au cours de l'épreuve, un candidat repère ce qui lui semble être une erreur d'énoncé, il le signale sur sa copie et poursuit sa composition en expliquant les raisons des initiatives qu'il a été amené à prendre.

La distance d'édition

Introduction

La distance d'édition, ou distance de Levensthein, donne une mesure de la similarité entre deux chaînes de caractères. Elle compte le nombre minimal d'opérations nécessaires pour passer d'une chaîne à l'autre, les opérations pouvant être de trois types :

— Insertion : ajouter un caractère.

— Suppression : supprimer un caractère.

— Substitution : modifier un caractère.

Par exemple on peut transformer la chaîne `"CAGTA"` en `"AGGTC"` par une suite de 3 opérations :

$$\texttt{CAGTA} \xrightarrow[\text{Suppression}]{} \texttt{AGTA} \xrightarrow[\text{Insertion}]{} \texttt{AGGTA} \xrightarrow[\text{Substitution}]{} \texttt{AGGTC}$$

et la distance d'édition de `"CAGTA"` à `"AGGTC"` est donc au plus 3 ; on pourrait se convaincre qu'elle est égale à 3 en vérifiant qu'on ne peut pas passer d'une chaîne à l'autre par moins de 3 opérations.

La distance d'édition a de nombreuses applications, parmi lesquelles :

– les correcteurs orthographiques l'utilisent pour suggérer des mots proches en cas d'erreur de frappe.

– mesurer la similarité entre deux séquences ADN en phylogénétique, chaque opération correspond alors à une mutation.

I. Implémentation des opérations

Les 3 opérations pouvant s'appliquer à une chaîne de caractère `mot` de longueur `n` sont :

- Insertion : on insère un caractère `c` à l'indice $i \in [\![0, n]\!]$. Par exemple une insertion du caractère `'*'` aux indices 0, 2 ou 4 dans la chaîne de caractère `'agro'`, donnerait :
  `'*agro'` `'ag*ro'` ou `'agro*'`
- Suppression : on supprime le caractère à l'indice $i \in [\![0, n-1]\!]$. Par exemple une suppression aux indices 0, 2 ou 3 dans la chaîne de caractère `'agro'`, donnerait :
  `'gro'` `'ago'` ou `'agr'`

- Substitution : on substitue un caractère c à l'indice $i \in [\![0, n-1]\!]$. Par exemple une substitution du caractère `'*'` aux indices 0, 2 ou 3 dans la chaîne de caractère `'agro'`,donnerait :

  `'*gro'` `'ag*o'` ou `'agr*'`

Écrivons les fonctions prenant en argument une chaîne de caractère `mot`, un entier `i` représentant l'indice i, et éventuellement le caractère c (c'est à dire une chaîne de caractère de longueur 1) et qui renvoient la chaîne obtenue après l'opération d'insertion, suppression ou substitution.
Les chaînes de caractère étant *non-mutables* on ne peut plus les modifer après création. Ainsi l'instruction `mot[i] = c` produirait une erreur. On créera plutôt une nouvelle chaîne de caractère en utilisant :

- un slicing : `mot[:i]` extrait le préfixe de longueur `i` de la chaîne `mot` ; il contient tous les caractères de `mot` de l'indice 0 inclus à l'indice `i` exclu. Par exemple : `'agro'[:2]` crée la nouvelle chaîne de caractère : `'ag'`
- un slicing : `mot[i:]` extrait le suffixe débutant à l'indice `i` de la chaîne `mot` ; il contient tous les caractères de `mot` à partir de l'indice `i` inclus. Par exemple : `'agro'[2:]` crée la nouvelle chaîne de caractère : `'ro'`
- la concaténation : `+` de deux chaînes de caractères ; elle crée une nouvelle chaîne de caractère obtenue en mettant bout à bout les deux chaînes. Par exemple `'agro' + '-' + 'veto'` crée la nouvelle chaîne de caractère `'agro-veto'`.

Le code de la fonction `substitution` est alors :

```
def substitution(mot,i,c):
    return mot[:i] + c + mot[i+1:]
```

Q 1.
a) Sur le même modèle, écrire le code de la fonction `suppression`.
b) Écrire de même le code de la fonction `insertion`

II. Résolution naïve par force brute

Pour calculer la distance d'édition, on pourrait envisager une approche naïve, consistant à appliquer à la première chaîne `mot1` toutes les opérations possibles afin d'obtenir toutes les chaînes à distance 1, puis de recommencer avec tous les mots ainsi obtenus pour obtenir les chaînes à distance au plus 2, et ainsi de suite, jusqu'à obtenir la deuxième chaîne `mot2` pour la première fois ; le nombre d'opérations éffectuées à partir de `mot1` pour obtenir `mot2` est alors la distance d'édition.
Les opérations d'insertion et de substitution n'ont besoin d'utiliser que les caractères figurant dans `mot2`. Aussi commençons par écrire une fonction `lettres` qui à partie d'une chaîne, renvoie la liste de tous les caractères figurant dans la chaîne, mais sans répétition. Par exemple appliqué à `'concours'` la fonction renverra la liste `['c','o','n','u','r','s']`.

```
1 def lettres(mot):
2     L = []
3     for car in mot:
4         if  ...        :    # à compléter
5             ...             # à compléter
6     return L
```

Q 2. Compléter les lignes 4 et 5 du code de la fonction `lettres` figurant ci-dessus.

La résolution naïve va alors procéder ainsi :
– On récupère au sein d'une liste `lettres_mot2` tous les caractères figurant dans la chaîne `mot2`.
– On initialise un dictionnaire `Dict` contenant le seul champ `0 : [mot1]`.
– Au sein d'une boucle perpétuelle on complète le dictionnaire : le champ de clé `d` contiendra la liste de tous les mots obtenus par une seule opération d'insertion, substitution ou suppression, à partir des chaines figurant dans la liste `Dict[d-1]` du champ de clé `d-1`. Ainsi `Dict[d]` sera la liste des mots obtenus de `mot1` par `d` opérations.
– Dès que `Dict[d]` contient `mot2` on arrête l'éxécution de la boucle pour renvoyer `d` ; c'est la valeur de la distance d'édition.

```
1  def force_brute(mot1,mot2):
2      Dict = {0:[mot1]}
3      d = 0
4      lettres_mot2 = lettres(mot2)
5      while True:
6          d += 1
7          Dict[d] = []
8          for mot in Dict[d-1]:
9              for i in range(len(mot)):
10                 Dict[d].append(    ...      )       # à compléter
11                 for car in lettres_mot2:
12                     Dict[d].append(    ...    )     # à compléter
13                     Dict[d].append(    ...    )     # à compléter
14             for car in lettres_mot2:
15                 Dict[d].append(    ...     )        # à compléter
16         if mot2 in Dict[d]:
17             return d
```

Q 3. Compléter les lignes 10, 12, 13 et 15 du code de la fonction `force_brute` figurant ci-dessus.

Le défaut de cette approche est que la taille du dictionnaire subit rapidement une inflation importante.

Q 4.
On suppose dans cette question que `mot1` et `mot2` sont les chaînes de caractères `'ACG'` et (respectivement) `'AGCTT'`.

a) Déterminer le nombre d'éléments de la liste `Dict[1]` dans ce cas.

Remarquons que les chaînes dans `Dict[1]` ne peuvent être que de longueurs 2, 3 ou 4.

b) Préciser le nombre d'éléments de la liste `Dict[1]` de longueurs 2, respectivement 3 et 4.

c) En déduire le nombre d'éléments de la liste `Dict[2]`.

Comme on le voit, la longueur des listes `Dict[d]` croît exponentiellement quand la valeur de `d` augmente, ce qui rend cette approche inenvisageable dans la pratique. Nous allons établir une autre approche, qui elle sera efficace.

III. Résolution par programmation dynamique

Considérons deux chaînes de caractères contenues dans des variables `mot1` et `mot2` ; on supposera que `mot1` est de longueur n et `mot2` est de longueur p.

Pour tout entiers $i \in [\![0, n]\!]$ et $j \in [\![0, p]\!]$ on considère :

• `mot1[:i]` préfixe de `mot1` de longueur i : si `mot1`= $a_0 a_1 \cdots a_{i-1} \cdots a_{n-1}$ alors `mot1[:i]`= $a_0 a_1 \cdots a_{i-1}$.
• `mot2[:j]` préfixe de `mot2` de longueur j : si `mot2`= $b_0 b_1 \cdots b_{j-1} \cdots b_{n-1}$ alors `mot2[:j]`= $b_0 b_1 \cdots b_{i-1}$.
• $D(i, j)$ est la distance d'édition de `mot1[:i]` et `mot2[:j]` c'est-à-dire le nombre minimal d'opérations

pour changer `mot1[:i]` en `mot2[:j]`. En particulier la distance d'édition de `mot1` à `mot2` est $D(n,p)$.

Le principe de résolution par programmation dynamique consiste à établir une relation de récurrence exprimant $D(n,p)$ à l'aide de valeurs de $D(i,j)$ pour des entiers $i < n$ et $j < p$.

- Pour tout $j \in [\![0,p]\!]$,
$$D(0,j) = j.$$
En effet $D(0,j)$ désigne la distance d'édition de la chaîne vide à `mot2[:j]`. Une suite d'opérations minimales ne peut consister qu'à insérer les j caractères de `mot2[:j]`.
- Pour tout $i \in [\![0,n]\!]$,
$$D(i,0) = i.$$
En effet où $D(i,0)$ désigne la distance d'édition de `mot1[:i]` à la chaîne vide. Une suite d'opérations minimales ne peut consister qu'à supprimer les i caractères de `mot1[:i]`.

Soit $i \in [\![1,n]\!]$ et $j \in [\![1,p]\!]$ établissons une relation de récurrence exprimant $D(i,j)$. Deux cas se présentent :

- Si `mot1[:i]` et `mot2[:j]` ont même dernier caractère, c'est-à-dire si `mot1[i-1]`= `mot2[j-1]`. Alors une suite d'opérations minimales transformant `mot1[:i]` en `mot2[:j]` s'obtient en laissant leur dernier caractère inchangé et en appliquant une suite d'opérations minimales transformant `mot1[:i-1]` en `mot2[:j-1]` ; ainsi :
$$D(i,j) = D(i-1,j-1).$$
- Si `mot1[:i]` et `mot2[:j]` n'ont pas même dernier caractère, c'est-à-dire si `mot1[i-1]`≠ `mot2[j-1]`. Alors une suite minimale d'opérations transformant `mot1[:i]` en `mot2[:j]` consistera soit :
  - en une suite minimale d'opérations transformant `mot1[:i]` en `mot2[:j-1]` avant d'insérer en dernière position le dernier caractère de `mot2[:j]`. Dans ce cas : $D(i,j) = D(i,j-1) + 1$.
  - en la suppression du dernier caractère de `mot1[:i]` puis en une suite minimale d'opérations transformant `mot1[:i-1]` en `mot2[:j]`. Dans ce cas : $D(i,j) = D(i-1,j) + 1$.
  - en la substitution du dernier caractère de `mot1[:i]` par le dernier caractère de `mot2[:j]` puis en une suite minimale d'opérations transformant `mot1[:i-1]` en `mot2[:j-1]`. Dans ce cas : $D(i,j) = D(i-1,j-1) + 1$.

  Ainsi par minimalité de la suite d'opérations :
$$D(i,j) = 1 + \min\big(D(i,j-1), D(i-1,j), D(i-1,j-1)\big)$$

Finalement on obtient la relation de récurrence :

$$D(i,j) = \begin{cases} j & \text{si } i = 0 \\ i & \text{si } j = 0 \\ D(i-1,j-1) & \text{si } i>0, j>0 \text{ et } \texttt{mot1[i-1]= mot2[j-1]} \\ 1 + \min\big(D(i,j-1), D(i-1,j), D(i-1,j-1)\big) & \text{si } i>0, j>0 \text{ et } \texttt{mot1[i-1]}\neq \texttt{mot2[j-1]} \end{cases}$$

Le calcul de $D(n,p)$ va alors s'effectuer au sein d'une matrice ayant $(n+1)$ ligne et $(p+1)$ colonnes dont l'élément ligne $i \in [\![0,n]\!]$ colonne $j \in [\![0,p]\!]$ représente $D(i,j)$ en commençant par remplir la première ligne et la première colonne, puis en remplissant ligne après ligne et colonne après colonne selon la relation de récurrence ci-dessus. Par exemple pour les chaînes de caractères `mot1='ACT'` et `mot2='AGTC'` la matrice D sera :

		A	G	T	C
	0	1	2	3	4
A	1	0	1	2	3
C	2	1	1	2	2
T	3	2	2	1	2

Q 5. Construire de même la matrice D lorsque `mot1='ACTCT'` et `mot2='AGTCAG'`.

La matrice D sera implémentée à l'aide d'une liste `D` constituée de $(n+1)$ listes de longueurs $(p+1)$ représentant les $(n+1)$ lignes de la matrice D.

On commencera par initialiser une matrice nulle avant de la compléter comme expliqué ci-dessus.

```
def matrice_nulle_1(n,p):
    return [[0]*(p+1) for k in range(n+1)]
```

```
def matrice_nulle_2(n,p):
    return [[0]*(p+1)] * (n+1)
```

```
def matrice_nulle_3(n,p):
    M =[ ]
    for i in range(p+1):
        L = []
        for j in range(n+1):
            L.append(0)
        M.append(L)
    return M
```

```
def matrice_nulle_4(n,p):
    M =[ ]
    for i in range(n+1):
        L = []
        for j in range(p+1):
            L.append(0)
        M.append(L)
    return M
```

Q 6. Parmi les 4 fonctions ci-dessus donner toutes les fonctions qui permettent d'initialiser une telle matrice nulle de $(n+1)$ lignes et $(p+1)$. On ne demande pas de justifier les réponses.

Dans la suite une telle fonction sera appelée `matrice_nulle`.

Q 7. Compléter les lignes 3, 5, 7, 11 et 13 du code de la fonction `matrice_levenshtein` ci-dessous prenant en paramètres deux chaînes `mot1` et `mot2` et qui renvoie la matrice D obtenue par la méthode expliquée dans cette partie.

```
1  def matrice_levenshtein(mot1,mot2):
2      n, p = len(mot1), len(mot2)
3      D = ...                          # à compléter
4      for j in range(p+1):
5          D[0][j] = ...                # à compléter
6      for i in range(n+1):
7          D[i][0] = ...                # à compléter
8      for i in range(1,n+1):
9          for j in range(1,p+1):
10             if mot1[i-1] == mot2[j-1]:
11                 D[i][j] = ...        # à compléter
12             else:
13                 D[i][j] = ...        # à compléter
14     return D
```

Q 8. En déduire le code d'une fonction `distance` prenant en paramètre deux chaînes de caractères `mot1` et `mot2` et qui renvoie leur distance d'édition.

IV. Application : correcteur orthographique

Un correcteur orthographique produira pour un mot mal orthographié tous les mots bien orthographiés qui en sont à distance d'édition minimale.

On suppose disposer pour cela d'une liste de tous les mots de la langue française :

```
Mots_Francais = [a, à, abaissa, abaissable, ..., zymotiques, zython, zythons, zythum]
```

Q 9. Compléter les lignes 6, 9 et 11 du code ci-dessous de la fonction `correcteur_orthographique` prenant en paramètre une chaîne de caractère `mot` et qui renvoie la liste de tous les mots bien orthographiés et dont la distance d'édition à `mot` est minimale.

```
1   def correcteur_orthographique(mot):
2       n = len(mot)
3       dmin = n+1
4       M = []
5       for chaine in Mots_Francais:
6           d = ...                 # à compléter
7           if d < dmin:
8               dmin = d
9               ...                 # à compléter
10          elif d == dmin:
11              ...                 # à compléter
12      return M
```

Par exemple, `correcteur_orthographique('mason')` renverra `['macon','maçon','maison','maton']`.

V. Obtention d'une suite d'opérations minimale

Les suites minimales d'opérations transformant une chaîne `mot1` en une chaîne `mot2` peuvent s'obtenir à partir de la matrice D décrite dans la partie III.

Pour cela il suffit de construire simultanément :

- un chemin : $(0,0), \dots\dots, (i,j), \dots\dots, (n,p)$ dans la matrice D, reliant $D_{0,0}$ à $D_{n,p}$
- une suite : $\texttt{mot1} = \texttt{mot}(0,0), \dots, \texttt{mot}(i,j), \dots, \texttt{mot}(n,p) = \texttt{mot2}$ de chaînes de caractères de `mot1` à `mot2`.

On procède 'à l'envers' en partant du point d'arrivée (n,p) et de la chaîne finale $\texttt{mot}(n,p)$ pour remonter jusqu'au point de départ $(0,0)$ et à la chaîne initiale. Soit $i \in [\![0,n]\!]$ et $j \in [\![0,p]\!]$; depuis l'élément (i,j) :

- si les chaînes `mot1[:i]` et `mot2[:j]` ont même dernier caractère (si `mot1[i-1]=mot2[j-1]`), alors on remonte à l'élément :

  ⑊ $(i-1, j-1)$; aucune opération sur la chaîne n'a été effectuée : $\texttt{mot}(i-1,j-1) = \texttt{mot}(i,j)$.

- si les chaînes `mot1[:i]` et `mot2[:j]` n'ont pas même dernier caractère (si `mot1[i-1]`≠`mot2[j-1]`), alors on remonte à l'élément :

  ↖ $(i-1, j-1)$ si $D(i-1,j-1) = D(i,j) - 1$.
  Dans ce cas $\texttt{mot}(i-1,j-1)$ s'obtient en substituant dans $\texttt{mot}(i,j)$ à l'indice $j-1$ le caractère `mot1[i-1]`.

  ← $(i, j-1)$ si $D(i,j-1) = D(i,j) - 1$.
  Dans ce cas $\texttt{mot}(i,j-1)$ s'obtient en supprimant dans $\texttt{mot}(i,j)$ le caractère d'indice $j-1$.

  ↑ $(i-1, j)$ si $D(i-1,j) = D(i,j) - 1$.
  Dans ce cas $\texttt{mot}(i-1,j)$ s'obtient en insérant dans $\texttt{mot}(i,j)$ à l'indice j le caractère `mot1[i-1]`.

  Lorsque plusieurs choix sont possibles chacun d'entre-eux aboutit à une suite minimale d'opérations.

Par exemple pour `mot1='ACT'` et `mot2='AGTC'` :

		A	G	T	C
	0	1	2	3	4
A	1	**0**	1	2	3
C	2	1	**1**	2	2
T	3	2	2	**1**	**2**

Donne pour suite minimale d'opérations : ACT $\longrightarrow$ AGT $\longrightarrow$ AGTC

Q 10. Construire par cette méthode toutes les suites minimales d'opérations changeant ACTCT en AGTCAG. La matrice D correspondante a été obtenue à la question 5.

```
1  def suite_operations(mot1,mot2):
2      D = ...                                  # à compléter
3      Suite_chgt = [mot2]
4      mot = mot2
5      i, j = len(mot1), len(mot2)
6      while i > 0 or j > 0:
7          if i>0 and j>0 and mot1[i-1] == mot2[j-1]:
8              i, j = i-1, j-1
9          elif i>0 and j>0 and D[i-1][j-1] == D[i][j]-1:
10             mot = ...                        # à compléter
11             i, j = i-1, j-1
12             Suite_chgt = [mot] + Suite_chgt
13         elif j > 0 and D[i][j-1] == D[i][j] -1:
14             mot = ...                        # à compléter
15             j = j-1
16             Suite_chgt = [mot] + Suite_chgt
17         elif i > 0 and D[i-1][j] == D[i][j]-1:
18             mot = ...                        # à compléter
19             Suite_chgt = [mot] + Suite_chgt
20             i = i-1
21     return Suite_chgt
```

Q 11. Compléter les lignes 2, 10, 14 et 18 du code ci-dessus de la fonction `suite_operations` prenant en argument deux chaînes de caractère et qui renvoie une suite minimale d'opérations changeant la première en la seconde par cette méthode.