Correcteur d'orthographe

Le problème suivant se compose de deux parties largement indépendantes ; dans la partie **II**, *seule la définition de la distance de Levenshtein, introduite dans la partie* **I**, *est nécessaire.*
Il est très vivement conseillé d'expliquer les codes que vous proposez, notamment le rôle des différentes variables que vous pouvez être amené à introduire.

Dans tout le sujet, on appelle « mot » une chaîne de caractères sans espace.

On rappelle quelques commandes sur les chaînes de caractères (rappel non exhaustif) : si `ch` est une chaîne de caractères

- `len(ch)` renvoie la longueur de la chaîne, ie le nombre de lettres qui la compose
- si $0 \leqslant$ `i` $< n =$ `len(ch)`, `ch[i]` renvoie la chaîne de caractère dont l'unique caractère est le $i^{\text{ème}}$ caractère de `ch`
- `ch[i:j]` renvoie la chaîne de caractères extraite de `ch` en ne conservant que les lettres de `ch` d'indices compris dans l'intervalle $[\![i, j-1]\!]$; cette chaîne sera vide si $i \geqslant$ `len(ch)` ou si $j \leqslant 0$ ou si $i \geqslant j$.
- le symbole `+` permet de concaténer les chaînes de caractères ; si $0 \leqslant i <$ `len(ch)`, pour remplacer le caractère d'indice i de `ch` par la lettre `a`, on peut utiliser `ch = ch[:i]+'a'+ch[i+1:]`

I Distance entre deux mots

1. Distance de Hamming

La distance de Hamming entre deux mots `ch1` et `ch2` de même longueur n, est le nombre d'indices $i \leqslant n-1$ tels que `ch1[i]` $\neq$ `ch2[i]`.

a) Quelle est la distance de Hamming entre `'pcsi'` et `'mpsi'` ?

b) Écrire une fonction `Hamming(ch1:str,ch2:str)->int` qui renvoie la distance de Hamming entre les mots `ch1` et `ch2`, supposés de même longueur.

c) Évaluez la complexité de votre fonction en fonction de la longueur n des deux mots `ch1` et `ch2`.

2. Distance de Levenshtein

Soient `ch1` et `ch2` deux mots de longueurs quelconques. On cherche à transformer `ch1` en `ch2` en n'utilisant que trois transformations élémentaires :

- la substitution d'un caractère de `ch1` par un autre caractère de l'alphabet
- l'insertion d'un caractère de l'alphabet à n'importe quel endroit dans `ch1`
- la suppression de n'importe lequel des caractères de `ch1`

La distance de Levenshtein (ou distance d'édition) entre `ch1` et `ch2` est le nombre minimal de transformations nécessaires pour passer de `ch1` à `ch2`.

Par exemple, la distance de Levenshtein de `'distance'` à `'édition'` est 6 (on peut vérifier qu'il est impossible de passer de `'distance'` à `'édition'` avec au plus 5 transformations) :

$$\texttt{'distance'} \overset{\text{supp}}{\to} \texttt{'distanc'} \overset{\text{supp}}{\to} \texttt{'distan'} \overset{\text{ins}}{\to} \texttt{'distaon'} \overset{\text{modif}}{\to} \texttt{'distion'} \overset{\text{supp}}{\to} \texttt{'dition'} \overset{\text{ins}}{\to} \texttt{'édition'}$$

a) Quelle est la distance de Levenshtein entre `'mpii'` et `'psi'` ?

On note, pour $0 \leqslant i \leqslant n =$ `len(ch1)` et $0 \leqslant j \leqslant m =$ `len(ch2)`, $d_{i,j}$ la distance de Levenshtein de `ch1[:i]` à `ch2[:j]`.

b) Justifier rapidement que

$$d_{i,j} = \begin{cases} i+j & \text{si } i = 0 \text{ ou } j = 0 \\ d_{i-1,j-1} & \text{si } i \neq 0, j \neq 0 \text{ et } \texttt{ch1[i-1]} = \texttt{ch2[j-1]} \\ 1 + \min\{d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}\} & \text{si } i \neq 0, j \neq 0 \text{ et } \texttt{ch1[i-1]} \neq \texttt{ch2[j-1]} \end{cases}$$

c) On considère la fonction suivante qui renvoie la distance de Levenshtein de `ch1` à `ch2` :

```python
def Lev(ch1:str,ch2:str) :
    n,m = len(ch1),len(ch2)
    if n*m == 0 :
        return n+m
    else :
        d1 = Lev(ch1[:n-1],ch2[:m-1])
        if ch1[n-1] == ch2[m-1] :
            return d1
        else :
            d2 = Lev(ch1,ch2[:m-1])
            d3 = Lev(ch1[:n-1],ch2)
            return 1+min(d1,d2,d3)
```

Expliquez pourquoi cette fonction est inefficace.

d) Écrire une fonction `tableauLEV(ch1:str,ch2:str)->list` qui prend en argument deux mots et qui renvoie le « tableau » `D` de taille $(n+1)\times(m+1)$ (sous forme d'une liste de $n+1$ sous listes de longueurs $m+1$) tel que `D[i][j]` $= d_{i,j}$ pour $0 \leqslant i \leqslant n =$ `len(ch1)` et $0 \leqslant j \leqslant m =$ `len(ch2)`.

e) En déduire une fonction `distLEV(ch1:str,ch2:str)->int` qui renvoie la distance de Levenshtein de `ch1` à `ch2`.

f) Évaluer la complexité de cette fonction en fonction de $n =$ `len(ch1)` et $m =$ `len(ch2)`.

g) On admet que le « tableau » renvoyé par `tableauLEV('distance','édition')` est la suivant :

0	1	2	3	4	5	6	7
1	1	1	2	3	4	5	6
2	2	2	1	2	3	4	5
3	3	3	2	2	3	4	5
4	4	4	3	2	3	4	5
5	5	5	4	3	3	4	5
6	6	6	5	4	4	4	4
7	7	7	6	5	5	5	5
8	8	8	7	6	6	6	6

Expliquer clairement comment on peut retrouver les manipulations à faire à partir de `'distance'` pour aboutir à `'édition'`.

h) Écrire une fonction `transfoLEV(ch1:str,ch2:str)->list` qui renvoie la liste des mots intermédiaires pour passer de `ch1` à `ch2`.
Par exemple `transfoLEV('distance','édition')` pourra renvoyer

```
['distance', 'distanc', 'distan', 'distaon', 'distion', 'dition', 'édition']
```

i) En vous inspirant de la fonction donnée à la question **I.2.c** et de la technique de mémoïsation avec un dictionnaire, donner une fonction `LEV(ch1:str,ch2:str)->int` faisant appel à une fonction récursive et qui renvoie la distance de Levenshtein de `ch1` à `ch2`.

II Correction d'un mot

Pour effectuer de la correction orthographique, on dispose d'un ensemble de mots valides (ex. issus du dictionnaire de la langue française). Cet ensemble est représenté sous la forme d'un arbre dont les arêtes sont étiquetées par des lettres de l'alphabet ; la fin d'un mot valide est marquée par une arête étiquetée par le symbole '$'.

L'arbre ci dessous est l'arbre associé aux mots `'mpsi','mpii','pcsi','mp','mpi','psi','pc'`.

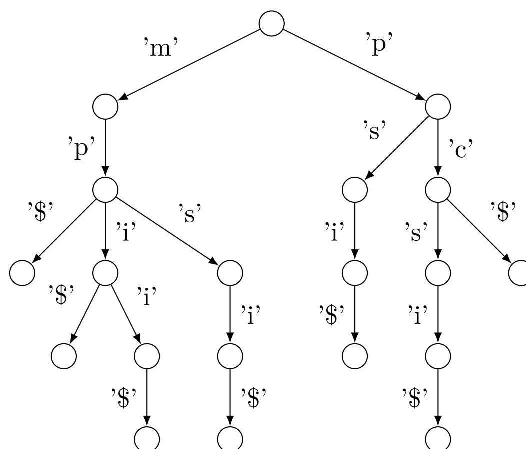


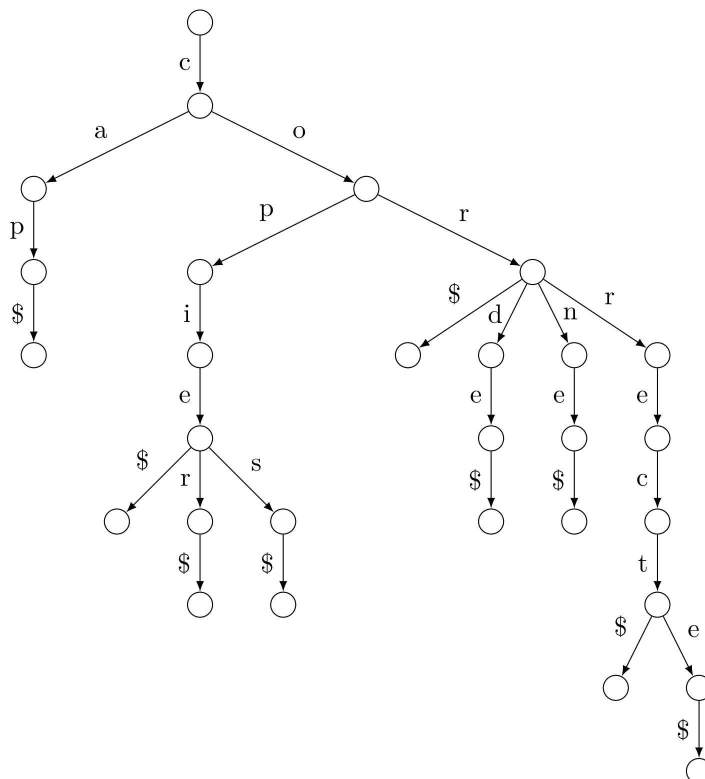
Fig 1 : les CPGE du lycée Montaigne

Un tel arbre est implémenté par un dictionnaire donc les clefs sont les différentes étiquettes (donc des chaînes de caractère de longueur 1) et les valeurs associées sont les sous arbres correspondants (donc un dictionnaire). Par exemple, l'arbre précédent est représenté par le dictionnaire

```
dico = {'m' :
          {'p' :
              {'$' : {} , 'i' :
                  {'$' : {} , 'i' : {'$' : {}}}
              ,'s' : {'i' : {'$' : {}}}
              }
          }
     ,'p' :
          {'c' :
              {'$' : {} , 's' :
                  {'i' : {'$' : {}}}
                  }
          ,'s' :
              {'i' : {'$' : {}}}
          }
     }
```

1. Fonctions sur les arbres

a) On considère l'arbre suivant :



Donner tous les mots associés à cet arbre.

b) Quel est la rôle de la fonction suivante, `dico` étant supposé être un dictionnaire du type décrit précédemment ?

```python
def f(dico) :
    n = 0
    for lettre in dico :
        if lettre == '$' :
            n += 1
        else :
            n += f(dico[lettre])
    return n
```

c) On souhaite coder une fonction <u>récursive</u> `contient(mot:str,dico:dict)->bool` qui renvoie `True` ou `False` selon qu'un mot `mot` appartient à un dictionnaire `dico`, du type précédent, ou non : compléter la fonction suivante

```
def contient(mot,dico) :
    if len(mot) == 0:
        return '$' in dico
    else :
        lettre = mot[0]
        if lettre in dico :
            mot = mot[1:]
            dico = dico[lettre]
            # return
        else :
            # return
```

d) Écrire une fonction `branche(mot:str)->dict` qui prend en argument un mot `mot` et qui renvoie le dictionnaire associé à l'arbre contenant uniquement le mot `mot`.

Par exemple `branche('abc')` devra renvoyer le dictionnaire `{'a': {'b': {'c': {'$': {}}}}}` alors que `branche(' ')` doit renvoyer `{'$':{}}`.

e) En déduire, en utilisant entre autres la fonction `branche`, une fonction `ajoutMot(mot:str,dico:dict)->None` qui prend en argument un mot `mot` et un dictionnaire `dico` représentant un arbre du type précédent et qui ajoute le mot `mot` à ce dictionnaire ; la fonction devra modifier le dictionnaire `dico` lui même.

On pourra s'inspirer de la structure de la fonction `contient` et il n'est pas nécessaire de vérifier avant si le mot `mot` fait déjà parti du dictionnaire.

2. Corriger une faute de frappe

Afin de corriger une éventuelle faute de frappe, on souhaite trouver, s'il existe, un mot du dictionnaire proche de celui qui a été saisi. On se limitera à la recherche d'un mot du dictionnaire dont la distance de Levenshtein (définie dans la partie **I**) avec le mot saisi est au plus 1.

a) En considérant le dictionnaire représenté par l'arbre donné au début de cette partie (Fig 1), donner, s'il en existe, un mot à distance de Levenshtien au plus 1 des mots suivants : `'pp'`, `'oc'`, `'ps'`, `'pps'` et `'psii'`.

b) Écrire une fonction `correctionLev(mot:str,dico:dict)` qui prend en argument un mot `mot` et un dictionnaire `dico` représentant un arbre du type précédent et qui renvoie un mot du dictionnaire donc la distance de Levenshtein avec le mot `mot` est au plus 1. Lorsque le mot `mot` est déjà un mot du dictionnaire, la fonction devra renvoyer `mot` lui même, elle devra renvoyer `' '` lorsque `mot` est déjà égal à `' '`, même si la chaîne `' '` n'est pas forcément dans le dictionnaire `dico` ; la fonction ne renverra rien (`None`) lorsqu'aucun mot du dictionnaire ne convient.