Devoir à la maison n°2 - Corrigé

1 Problème 1

1.1 Programmation récursive

1. (a) Si s (ou t) est la chaîne vide, alors `longueur(s,t)=0`.

   (b) Notons `n=len(s)` et `p=len(t)`.
   Si `s[n-1]=t[p-1]`, alors alors `longueur(s,t)=1+longueur(s[:n-1],t[:p-1])` (car la dernière lettre est commune aux deux mots, et on cherche alors la plus grande séquence commune sur les lettres d'avant)

   (c) Si `s[n-1]!=t[p-1]`, alors `longueur(s,t)=MAX(longueur(s[:n-1],t),longueur(s,t[:p-1])`.

   (d) Code de la fonction `longueur` de façon récursive avec mémoïsation (cette fonction retourne un nombre) :

```
1  D={}
2
3  def longueur(s,t):
4      if (s,t) in D:
5          return D[(s,t)]
6      n=len(s)
7      p=len(t)
8      if n==0 or p==0:
9          D[(s,t)]=0
10         return 0
11     if s[n-1]==t[p-1]:
12         D[(s,t)]=longueur(s[:n-1],t[:p-1])+1
13         return D[(s,t)]
14     else:
15         D[(s,t)]=max([longueur(s[:n-1],t),longueur(s,t[:p-1])])
16         return D[(s,t)]
```

2. Code de la fonction `PLSC(s,t)` de façon récursive avec mémoïsation (cette fonction retourne une chaine de caractères) :

```
D2={}
#clef: un tuple de 2 mots
#valeur: une liste avec la longueur de la plus grande séquence commune, et la valeur de cette séquence

def PLSC(s,t):
    if (s,t) in D2:
        return D2[(s,t)]
    n=len(s)
    p=len(t)
    if n==0 or p==0:
        D2[(s,t)]=[0,'']
        return D2[(s,t)]
    if s[n-1]==t[p-1]:
        couple=PLSC(s[:n-1],t[:p-1])
        D2[(s,t)]=[couple[0]+1,couple[1]+s[n-1]]
        return D2[(s,t)]
    else:
        couple1=PLSC(s[:n-1],t)
        couple2=PLSC(s,t[:p-1])
        if couple1[0]>couple2[0]:
            D2[(s,t)]=[couple1[0], couple1[1]]
        else:
            D2[(s,t)]=[couple2[0], couple2[1]]
        return D2[(s,t)]
```

3. Nous allons tout d'abord récupérer dans une liste tous les mots contenus dans le fichier, puis nous les comparerons deux à deux pour trouver la plus grande séquence commune en tre ces deux mots. Nous retiendrons la plus grande.

```
f=open('liste_mots.txt','r')
liste=f.readlines()
print('Il y a ', len(liste), ' mots dans le fichier')
f.close()

#on va nettoyer la liste (= enlever les \n)
n=len(liste)
for i in range(n):
    liste[i]=liste[i].strip()


maxi_longueur=-np.inf

for i in range(n):
    for j in range(i+1,n): #attention à ne pas comparer un mot avec lui-même
        valeur=longueur(liste[i].strip(),liste[j].strip())
        if valeur>maxi_longueur:
            maxi_longueur=valeur
            couple=[liste[i],liste[j]]

print(couple, PLSC(couple[0],couple[1]))
```

Le résultat est le suivant : ['appartement', 'parfaitement'] [9, 'partement']

1.2 Programmation itérative

4. Nous allons créer le tableau contenant les longueurs cherchées, en le remplissant petit à petit :

```
def longueur2(s,t):
    n=len(s)
    m=len(t)
    tab=[[0 for j in range(m)]for i in range(n)]
    #tab[i][j] contiendra la longueur de la plus longue séquence commune
    # entre s[:i+1] et t[:j+1]

    if s[0]==t[0]:
        tab[0][0]=1

    #remplissage de la première ligne
    for j in range(1,m):
        if s[0]==t[j]:
            tab[0][j]=1
        else:
            tab[0][j]=max([0,tab[0][j-1]])

    #remplissage de la première colonne
    for i in range(1,n):
        if s[i]==t[0]:
            tab[i][0]=1
        else:
            tab[i][0]=max([0,tab[i-1][0]])

    #remplissage des autres cases de proche en proche
    for i in range(1,n):
        for j in range(1,m):
            if s[i]==t[j]:
                tab[i][j]=1+tab[i-1][j-1]
            else:
                tab[i][j]=max([tab[i-1][j],tab[i][j-1]])

    return tab[-1][-1]
```

---

2 Problème 2 : Pour s'entraîner à manipuler les listes et les dictionnaires

Exercice 1 : .

```
jours={"lundi":1,"mardi":2,"mercredi":3,"jeudi":4,"vendredi": 5,"samedi":6,"dimanche":8}
jours["dimanche"]=7

#affichage de la liste des clefs
 liste_clefs =[]
for i in jours:
      liste_clefs .append(i)
print( liste_clefs )
```

```
9
10 #affichage de la liste des valeurs
11 liste_val =[]
12 for i in jours:
13     liste_val.append(jours[i])
14 print(liste_val)
15
16 #affichage de la liste des paires
17 liste_paires =[]
18 for i in jours:
19     liste_paires.append((i,jours[i]))
20 print(liste_paires)
```

Exercice 2 : .

```
1 def concatener(dico1,dico2):
2     d={}
3     for i in dico1:
4         d[i]=dico1[i]
5     for i in dico2:
6         d[i]=dico2[i]
7     return d
```

Exercice 3 : .

```
1 def partition(dico):
2     d1={}
3     d2={}
4     for i in dico:
5         if d[i]>=10:
6             d1[i]=d[i]
7         else:
8             d2[i]=d[i]
9     return d1,d2
```

Exercice 4 : .

```
1 def parite(liste):
2     d={}
3     for n in liste:
4         if n%2==0:
5             d[n]='pair'
6         else:
7             d[n]='impair'
8     return d
```

Exercice 5 : .

```
1 def comptage(mot):
2     d={}
3     for car in mot:
4         if car in d:
5             d[car]+=1
6         else:
7             d[car]=1
8     return d
```

Exercice 6 : .

```
def recherche_inv(d,v):
    for i in d:
        if d[i]==v:
            return i
    return None  #cas où la valeur v n'a pas été trouvée

def recherche_inv_liste(d,v):
    liste =[]
    for i in d:
        if d[i]==v:
            liste.append(i)
    return liste
```

3 Problème 3 : Chemin dans un labyrinthe

Importation des modules nécessaires :

```
import random as rd
import numpy as np
import matplotlib.pyplot as plt
```

1. *Création d'un labyrinthe :*

```
def creation_lab(n,p):
    lab=np.zeros((n,n)) #que des murs au départ
    for i in range(n):
        for j in range(n):
            nb=rd.random() #un nombre au hasard entre 0 et 1
            if nb<p: #ce nombre est inférieur à p avec une probabilité p
                lab[i,j]=1 # création d'un passage
    return lab
```

2. *Affichage d'un labyrinthe :*

```
def affiche(lab):
    plt.imshow(lab,cmap='grey')
```

3. *Existence d'un chemin entre deux cellules :*
   Nous allons d'abord écrire une fonction **`voisins(lab,x,y)`** qui retourne la lsite des ces voisines de (x, y)

```
def voisins(lab,x,y):
    ''' retourne la liste des cases accessibles voisines '''
    n=len(lab) #la taille du côté du labyrinthe
    liste =[]
    for coord in [[x-1,y],[x,y-1],[x,y+1],[x+1,y]]:

        if coord[0]>-1 and coord[0]<n and coord[1]>-1 and coord[1]<n and lab[coord[0],coord[1]]==1:
            liste.append(coord)
    return liste
```

```python
def existe_chemin(lab,x1,y1,x2,y2):
    if lab[x1,y1]==0 or lab[x2,y2]==0:
        return False
    pile=[[x1,y1]]
    n=len(lab) #la taille du côté du labyrinthe
    marquage=np.array([[False for i in range(n)]for i in range(n)])
    while len(pile)!=0:
        #print(pile)
        c=pile.pop()
        if marquage[c[0],c[1]]==False: #si on n'a pas encore visité cette case
            if c==[x2,y2]:
                return True #on atteint la case souhaitée
            marquage[c[0],c[1]]=True
            L=voisins(lab,c[0],c[1])
            for v in L: #on parcourt les voisins de c
                if marquage[v[0],v[1]]==False:
                    pile.append(v)
    return False
```

4. *Complexité :*
   Si on note n la taille du labyrinthe, alors la complexité la fonction `existe_chemin` est en $O(n^2)$. En effet, la boucle $while$ est en $O(n^n 2)$ dans le pire des cas, (n^2 cases possibles dans la pile). La boucle for qui à l'intérieur tourne au maximum 4 fois (le nombre maximum de voisins d'une case), et le reste des instructions est de complexité constante.
5. *Obtention d'un chemin entre deux cellules :*

```python

def chemin(lab,x1,y1,x2,y2):
    if lab[x1,y1]==0 or lab[x2,y2]==0:
        return False
    pile=[[x1,y1]]
    n=len(lab) #la taille du côté du labyrinthe
    marquage=np.array([[False for i in range(n)]for i in range(n)])
    predecesseur=np.array([[None for i in range(n)]for i in range(n)])
    while len(pile)!=0:
        #print(pile)
        c=pile.pop()

        if marquage[c[0],c[1]]==False: #si on n'a pas encore visité cette case
            if c==[x2,y2]: #on atteint la case souhaitée
                liste_envers=[c]
                while c !=[x1,y1]:
                    c=predecesseur[c[0],c[1]]
                    liste_envers.append(c)
                return liste_envers[::-1]
            marquage[c[0],c[1]]=True
            L=voisins(lab,c[0],c[1])
            for v in L: #on parcourt les voisins de c
                if marquage[v[0],v[1]]==False:
                    predecesseur[v[0],v[1]]=c
                    pile.append(v)
    return False
```

6. Pour améliorer le parcours, on peut utiliser un parcours en largeur, car les cellules seront alors explorées "génération par génération".