Proposition de corrigé du TP2 : Listes, récursivité, tris

1 Listes

Exercice TP2.1 *Recherche dans une liste ou un tableau*

1. On utilise ici la méthode du candidat :

```
def maximum(L):
    candidat=L[0]
    for k in range(len(L)):
        if L[k] > candidat:
                candidat=L[k]
    return(candidat)
```

2. On utilise une boucle conditionnelle conformément à l'énoncé

```
def appartient(L,x):
    n,k=len(L),0
    while k < len(L) and L[k]!=x:
        k+=1
    return(k!=n)
```

Exercice TP2.2 *Dictionnaires*

1. On utilise la méthode `keys` et on procède comme dans le cours.

```
def occurences(L):
    occ={}
    for i in L:
        if i in occ.keys():
            occ[i]+=1
        else :
            occ[i]=1
    return occ
```

2. On utilise simplement la fonction précédente et on convertit en liste.

```
def sansdoublon(L):
    occ=occurences(L)
    return list(occ.keys())
```

3. Les doublons correspondent à un nombre d'apparitions strictement plus grand que un et on utilise une boucle `for` comme précédemment.

```
def doublons(L):
    occ=occurences(L)
    M=[]
    for i in occ.keys():
        if occ[i]>1:
            M.append(i)
    return M
```

4. On procède par méthode du candidat pour obtenir la plus grande occurence.


```
def majoritaire(L):
    occ=occurences(L)
    m=L[0]
    n=occ[m]
    for i in occ.keys():
        if occ[i]>n:
            m=i
            n=occ[m]
    return m
```

2 Récursivité

Exercice TP2.3 *Récursivité : PGCD*

1. On donne une version impérative de l'algorithme d'Euclide en utilisant la définition de l'énoncé :

```
def PGCD(a,b):
    p,q = a,b
    while q!=0:
        p,q =q,p%q
    return p
```

puis une version récursive :

```
def PGCD(a,b):
    if b==0:
        return a
    else:
        return PGCD(b,a%b)
```

2. On modifie la fonction impérative précédente pour obtenir :

```
def decompte(a, b):
    c = 0
    p,q = a,b
    while q != 0 :
        p,q = q,p%q
        c += 1
    return c
```

Exercice TP2.4 *Récursivité : persistance*

1. On donne d'abord différentes version pour la fonction `prod`

```
def prod (n) :   # version impérative arithmétique
    m = n
    p = 1
    while m > 0 :
        p = p * (m % 10)
        m = m // 10
    return p

def prod (n) :   # version impérative sur caractères
    m = str (n)
    p = 1
    for c in m :
        p = p * int (c)
    return p
```

```
def prod (n) :   # version récursive
    if (n // 10) == 0 :
        return n
    else :
        return (n % 10) * (prod (n//10))
```

La fonction `persistance` en découle :

```
def persistance (n) :  # version impérative
    m = n
    p = 0
    while m > 9 :
        p += 1
        m = prod (m)
    return p

def persistance (n) :  # version récursive
    if (n // 10) == 0 :
        return 0
    else :
        return 1 + persistance (prod (n))
```

2. On trouve le premier nombre dont la persistance dépasse 4 :

```
>>> n = 100
>>> while persistance (n) < 5 :
        n += 1
        print (n)

>>> 679
```

3 Tris

Exercice TP2.5

1. Notons n la longueur de `L` et $p = (b-a)/n$. Alors un élément e de `L` est dans le k-ième intervalle si et seulement si $a + k * p \leq e < a + (k+1) * p$ c'est-à-dire si et seulement si $k \leq (e-a)/p < k+1$. On en déduit que le k recherché est égal à la partie entière de $(e-a)/p$.
2. En utilisant le tri fusion pour trier chaque paquets :

```
def triPaquets(L,a,b):
    n=len(L)
    I=[[] for k in range(n)]
    for x in L:
        k=int(n*(x-a)/(b-a))
        I[k]+=[x]
    LL=[]
    for k in range(n):
        I[k]=triFusion(I[k])
        LL+=I[k]
    return LL
```

3. La première boucle `for` s'exécute en $O(n)$. Dans le pire des cas, il y a un paquet de longueur n dont le tri s'effectuera en $O(n\ln(n))$. La complexité est alors en $O(n\ln(n))$. Dans le meilleur des cas, il n'y a qu'une seule valeur par parquet et la deuxième boucle `for` s'exécute en $O(n)$. La complexité est alors en $O(n)$.

Exercice TP2.6 *Tri à bulles*

1. On utilise la définition de ce tri par l'énoncé :

```
def tri_bulles(L):
    n = len(L)
    for a in range(1,n) :
        for k in range (0,n-a):
            if L[k] > L[k+1]:
                L[k] , L[k+1] = L[k+1] , L[k]
```

On peut également améliorer l'efficacité du programme en l'arrêtant dès qu'il n'y a plus d'échange effectué.

```
def tri_bulles2(L):
    n = len(L)
    for a in range(0,n-1) :
        stop=0
        for k in range (0,n-1-a):
            if L[k] > L[k+1]:
                L[k], L[k+1] = L[k+1], L[k]
                stop+=1
        if stop==0:
            return(None)
```

2. On utilise une boucle `for` pour concaténer les éléments successivement.

```
from random import randint

def liste_alea(n,N):
    liste_retour=[]
    for k in range(n):
        x = randint(1,N)
        liste_retour.append(x)
    return(liste_retour)
```

On peut également utiliser la fonction `choice`

```
def alea(N,n):
    return [choice(list(range(1,N))) for k in range(n)]
```

3. Dans le code suivant, on fait une moyenne temporelle sur 100 tirages aléatoires de listes.

```
import time
N = 1000000
nombre_repetitions = 100
liste_temps = []
n_max = 100
for n in range(1,n_max):
    T1 = time.time()
    for k in range(nombre_repetitions):
        tri_bulles(liste_alea(n,N))
    T2=time.time()
    liste_temps.append((T2-T1)/100)

import matplotlib.pyplot as plt

plt.plot(range(1,n_max),liste_temps)
plt.show()
```