RÉVISIONS Python 2 : Listes et gloutonnerie

Correction

I] Algorithmes sur les listes

Q1. Écrire une fonction `est_dedans(t,x)` qui prend en entrée une liste `t` et une valeurs `x` et qui renvoie `True` si `x` est un élément de `t` (`False` sinon). On n'utiliser pas l'expression `x in t`.

Correction : On parcoure la liste et dès qu'on voit l'élément on retourne `True`, si à la fin du parcours on ne l'a pas vu, c'est qu'il n'est pas présent. La complexité est en $O(n)$ (où n est la longueur de la liste).

Code Python :

```
def est_dedans(t,x):
    for i in range(0,len(t)):
        if t[i]==x:
            return True
    return False
```

Q2. Écrivez une fonction qui détermine le maximum d'une liste, une autre qui détermine la position de ce maximum (Bien entendu on n'utilisera pas la fonction `max`).
Et si le maximum est pris plusieurs fois dans la liste, lequel est renvoyé par cette seconde fonction ? Comment changer ce comportement ?
Élaborer un algorithme permettant de déterminer les deux plus grands éléments d'une liste, enfin l'implémenter en Python et le tester sur des exemples judicieusement choisis.

Correction : Le premier élément est le candidat à être le maximum, puis on parcours la liste et on ré-ajuste le maximum (éventuellement on garde en mémoire sa position), pour les deux plus grands, on commence par considérer les deux premiers comme les deux plus grands éléments (et on garde l'ordre) puis on parcours la liste et on réajuste. Tous ces algorithmes ont une complexité en $O(n)$.

Dans notre exemple on retourne la position de la première occurrence du plus grand élément, pour changer le comportement il suffit de mettre `if t[k]>=tmax`.

Code Python :

```
def maxi(t):
    tmax=t[0]
    n=len(t)
    for k in range(1,n):
        if t[k]>tmax:
            tmax=t[k]
    return tmax

def pos_maxi(t):
    tmax=t[0]
    tpos=0
    n=len(t)
    for k in range(1,n):
        if t[k]>tmax:
            tmax=t[k]
            tpos=k
    return tpos

def lesdeuxplusgrand(t):
    if t[0]>=t[1]:
        max1=t[0]
        max2=t[1]
    else:
        max1=t[1]
```

```
        max2=t[0]
    for i in range(2,len(t)):
        if t[i]>max2:
            if t[i]>max1:
                max2=max1
                max1=t[i]
            else:
                max2=t[i]
    return max1,max2
```

Q3. Écrivez une fonction prenant une liste en entrée et renvoyant le booléen *True* si la liste est croissante (*False* sinon).

Correction :
La liste est croissante si pour tout k on a t[k]<=t[k+1], donc elle n'est pas croissante s'il existe un k tel que t[k]>t[k+1], c'est ce qu'on fait
Code Python :

```
def est_croissante(t):
    res=True
    n=len(t)
    for k in range(n-1):
        if t[k] > t[k+1]:
            res=False
    return res
```

Q4. Écrivez un programme prenant en entrée une liste, et renvoyant la liste des sommes cumulées (depuis le premier terme). La complexité devra être en $O(n)$ où n est le nombre d'éléments dans la liste.

Correction :
Code Python :

```
def cumsomme(t):
    n=len(t)
    res=[t[0]]
    for k in range(1,n):
        res.append(res[k-1]+t[k])
    return res
```

Q5. Élaborer un algorithme permettant de rechercher les deux valeurs les plus proches dans une liste.

Correction : L'idée est assez proche, mais plus technique, de la question d'avant, on commence par déterminer l'élément le plus proche du premier, c'est notre couple candidat, ensuite pour tous les éléments entre le deuxième et l'avant dernier on détermine l'élément le plus proche qui se situe à droite (inutile à gauche car déjà traité), une foi qu'on a cela on compare avec notre couple candidat et on met à jours si besoin.
On a deux boucles imbriquées, ce qui donne une complexité en $O(n^2)$ (pour être plus précis le nombre d'opération est $2+5n-2+\sum_{i=1}^{n-2} 3+\sum_{k=i+2}^{n-1} 5$ si on compte seulement les additions, comparaisons et utilisation d'abs).
Code Python :

```
def deuxplusproches(t):
    c1=t[0]
    c2=t[1]
    d=abs(c1-c2)
    for k in range(2,len(t)):
        if abs(t[k]-c1)<d:
            c2=t[k]
            d=abs(c1-c2)
```

```
    # A ce stade c2 est la valeur la plus proche de t[0]

    for i in range(1,len(t)-1):
        ct1=t[i]
        ct2=t[i+1]
        dt=abs(ct1-ct2)
        for k in range(i+2,len(t)):
            if abs(t[k]-ct1)<dt:
                ct2=t[k]
                dt=abs(t[k]-ct1)
        if dt<d:
            c1=ct1
            c2=ct2
            d=dt
    return c1,c2
```

Q6. Écrire une fonction qui permet de trier une liste. Le tri pourra être en place ou pas, l'algorithme choisi est laissé libre : tri par insertion, tri fusion ou tri rapide.
Remarque : il en existe plein d'autres : tri à bulle, tri par sélection, et celui utilisé par par la fonction `sorted` de Python est le timsort inventé par Tim Peter en 2002 et qui est un dérivé du tri fusion et du tri par insertion et qui part du principe que dans un ensemble de données une parti est déjà triée, sa complexité est en $O(N)$ dans le meilleur des cas et en $O(N\ln(N))$ dans le pire et aussi en moyenne.

Correction : Notons N le nombre d'éléments de la liste.

Tri par insertion : On considère que la liste est triée entre L[0] et L[n-1] et on place L[n] à la bonne place (pour cela on décale les éléments plus grand que lui vers la droite), il est en $O(N^2)$.

Tri fusion : On coupe la liste en deux, on appel récursivement l'algo de trie sur les deux moitiés (si la liste a 1 élément il est trié) et on fusionne les deux listes triées, il est en $O(N\ln(N))$.

Tri rapide : Si la liste est vide (ou a un élément) alors elle est déjà triée, sinon on choisit un élément de la liste qu'on appelle pivot (par exemple le premier) et on considère la liste des éléments plus petits que le pivot et la liste des éléments plus grands, on trie les deux listes de la même manière et on retourne la liste triée. Il est possible de le faire en place mais c'est plus technique. La complexité est en $O(N^2)$ dans le pire des cas (le pivot est une extrémité) et en $O(N\ln(N))$ en moyenne

Remarque :
Tri à bulle : on compare les deux premiers éléments et on les ordonnes si besoin, puis entre le deuxième et le troisième et ainsi de suite jusque la fin, et on recommence un certain nombre de fois (les éléments plus grands "montent", comme une bulle dans un liquide.
Tri par sélection : on cherche le plus petit et on le met en premier puis on ré-itère,

Code Python :

```
##
## Tri par insertion
##

def tri_insertion_enplace(L):
    N = len(L)
    for n in range(1,N):
        cle = L[n]
        j = n-1
        while j>=0 and L[j] > cle:
            L[j+1] = L[j] # decalage
            j = j-1
        L[j+1] = cle

def tri_insertion(liste):
    L = list(liste) # copie de la liste
    N = len(L)
    for n in range(1,N):
        cle = L[n]
        j = n-1
        while j>=0 and L[j] > cle:
```

```
            L[j+1] = L[j] # decalage
            j = j-1
        L[j+1] = cle
    return L

##
## Tri fusion
##

def fusion(L1,L2):
    n1 = len(L1)
    n2 = len(L2)
    L12 = [0]*(n1+n2)
    i1 = 0
    i2 = 0
    i = 0
    while i1<n1 and i2<n2:
        if L1[i1] < L2[i2]:
            L12[i] = L1[i1]
            i1 += 1
        else:
            L12[i] = L2[i2]
            i2 += 1
        i += 1
    while i1<n1:
            L12[i] = L1[i1]
            i1 += 1
            i += 1
    while i2<n2:
            L12[i] = L2[i2]
            i2 += 1
            i += 1
    return L12


def tri_fusion(L):
    N=len(L)
    if N<=1:
        return L[:]
    else:
        milieu=len(liste)//2
        L1=tri_fusion(L[:milieu])
        L2=tri_fusion(L[milieu:])
        return fusion(L1,L2)
##
## Tri rapide
##

def tri_rapide(liste):
    if liste==[]:
        return liste
    liste_g=[]
    liste_d=[]
    for i in range(1,len(liste)):
        if liste[i]<=liste[0]:
            liste_g.append(liste[i])
        else:
            liste_d.append(liste[i])
    return tri_rapide(liste_g)+[liste[0]]+tri_rapide(liste_d)

##
## Tri rapide en place
```

```python
##

def partition(liste,g,d):
    pivot=liste[g]
    i=g+1
    j=d
    while i<=j:
        while i<len(liste) and liste[i]<=pivot:
            i=i+1
        while liste[j]>pivot:
            j=j-1
        if i<j:
            liste[i],liste[j]=liste[j],liste[i]
            i=i+1
            j=j-1
    liste[g],liste[j]=liste[j],liste[g]
    return j

def tri_rapide(liste,g,d):
    if g<d:
        j=partition(liste,g,d)
        tri_rapide(liste,g,j-1)
        tri_rapide(liste,j+1,d)
    return liste
```

II] Mastermind

L'objectif est de programmer des fonctions liées à un jeu de Mastermind, pour rappel c'est un jeu ou l'un des joueurs, appelé codeur, décide d'un code en choisissant 4 pions de couleurs (il y a 6 couleurs, et il peut en prendre plusieurs de la même couleur), on représentera ce choix par une liste de quatre entiers, chacun compris entre 0 et 5, chaque entier représentera une couleur, par exemple `[1,2,4,1]`. L'objectif du second joueur, appelé le décodeur, est de découvrir le code, pour cela il fera une proposition, par exemple `[1,1,2,3]`), ensuite le codeur lui indiquera le nombre de pions bien placés (ici il n'y en a qu'un) et le nombre de pions de la bonne couleur mais mal placé (ici il y en a deux), le décodeur peut ensuite faire une autre proposition, l'objectif est de découvrir le code en un minimum de tentatives (sans dépasser les 10 tentatives).

Q7. Écrire une fonction `occurrences(lst)` qui prend en argument une liste `lst` qu'on suppose composée d'entiers compris entre 0 et 5 et qui retourne une liste de 6 entiers, le premier sera le nombre d'occurrence de 0 dans `lst`, le second celui de de 1 dans `lst`, etc. Par exemple `occurences([1,2,4,1])` devra renvoyer `[0, 2, 1, 0, 1, 0]`

Correction :
Code Python :

```python
code=[1,2,4,1]
prop=[1,1,2,3]

def occurrences(lst):
    res=[0]*6
    for p in lst:
        res[p]+=1
    return res
print(occurrences(code))
```

Q8. Écrire une fonction `bien_places(lst1,lst2)` qui prend en argument deux listes et qui retourne le nombre de pions bien placés, par exemple `bien_places([1,2,4,1],[1,1,2,3])` doit renvoyer 1.

Correction :
Code Python :

```
def bien_places(lst1,lst2):
    res=0
    for k in range(len(lst1)):
        if lst1[k]==lst2[k]:
            res+=1
    return res
print(bien_places(code,prop))
```

Q9. Écrire une fonction `mal_places(lst1,lst2)` qui prend en argument deux listes et qui retourne le nombre de pions mal placés (de la bonne couleur, mais pas au bon endroit), par exemple `mal_places([1,2,4,1],[1,1,2,3])` doit renvoyer 2.
Indication : si on note $ol1_i$ (resp. $ol2_i$) le nombre d'occurrences de la couleur i dans la liste `lst1` (resp. `lst2`), alors $\min(ol1_i, ol2_i)$ est le nombre de pions de la couleurs i qui sont bien ou mal placés, ainsi le nombre de pions mal placés n'est rien d'autre que $\left(\sum_{i=0}^{5} \min(ol1_i, ol2_i)\right) - b$, où on a noté b le nombre de pions bien placés.

Correction :
Code Python :

```
def mal_places(lst1,lst2):
    res=0
    ol1=occurrences(lst1)
    ol2=occurrences(lst2)
    for k in range(len(ol1)):
        res+=min(ol1[k],ol2[k])
    res=res-bien_places(lst1,lst2)
    return res
print(mal_places(code,prop))
```

Q10. S'il vous reste du temps vous pouvez programmer un jeu de mastermind complet où l'ordinateur est le codeur, ou mieux un jeu ou c'est l'ordinateur le décodeur. Pour ce dernier il pourra être utile d'écrire une fonction qui détermine la liste de tous les codes possibles.

Correction :
Code Python :

```
def crea_lst_prop(n,ncoul):
    if n==1:
        return [ [k] for k in range(ncoul)]
    else:
        res=[]
        resm1=crea_lst_prop(n-1,ncoul)
        for lst in resm1:
            for k in range(ncoul):
                res.append(lst+[k])
        return res

print(crea_lst_prop(2,3))

def trouve(code):
    nprop=0
    infos=[]
    lst_prop=crea_lst_prop(4,6)

    prop=lst_prop[0]
    bp=bien_places(code,prop)
    mp=bien_places(code,prop)
    nprop+=1
    if bp==4:
        return nprop
    infos.append([prop,bp,mp])
```

```
        for k in range(1,len(lst_prop)):
            prop=lst_prop[k]
            a_tester=True
            for prec in infos:
                pprop,pbp,pmp=prec
                bp=bien_places(prop,pprop)
                mp=bien_places(prop,pprop)
                if bp!=pbp or mp!=pmp:
                    a_tester=False
            if a_tester:
                bp=bien_places(code,prop)
                mp=bien_places(code,prop)
                nprop+=1
                if bp==4:
                    return nprop,infos
                infos.append([prop,bp,mp])
    print(trouve(code))
```

III] Algorithme glouton

Un algorithme est dit glouton lorsqu'il résout un problème d'optimisation en une succession d'étapes et où à chaque étape il choisit la solution optimale, cependant il n'y a aucune garantie pour que la solution finale soit optimale. Par exemple si on désire dans une rue visiter toutes les boutiques en minimisant la distance parcourue un algorithme glouton consisterait à choisir à chaque étape la boutique la plus proche, il est facile de voir que dans ce cas là, la solution n'est pas nécessairement optimale (par exemple si on commence à la deuxième boutique de la rue, que la plus proche est la troisième etc. on devra à la fin revenir à la première boutique qu'on a pas visitée).

Q11. Un exemple classique est celui du rendu de monnaie, on considère qu'on a des pièces (et des billets) de 1,2,5,10,50 et 100 euros et qu'on doit rendre la monnaie en utilisant un minimum de pièce, si on doit rendre 42 euros on va clairement rendre 4 billets de 10 et une pièce de 2, quand on fait cela on applique un algorithme glouton : on choisit la plus grande valeur de la monnaie inférieure à 42, etc.

(a) Écrire une fonction `pieces_a_rendre(somme_a_rendre , systeme_monnaie)` qui prend en entrée la somme à rendre et la liste des pièces possible et qui retourne la liste des pièces à rendre.

(b) Que peut-on dire de l'algorithme pour un système de monnaie qui a des pièces de 1, 3 et 4 (on peut aussi considérer le système anglais d'avant sa réforme en 1971 [1]).

Correction : Pour (b) : Pour ce système l'algorithme glouton donne pour rendre 6 : 4, 1, 1 ; ce qui n'est pas optimal (c'est 3 et 3)

Code Python :

```
def pieces_a_rendre(somme_a_rendre , systeme_monnaie):
    lst_pieces = []
    # indice de la plus grosse pièce du système de monnaie
    i = len(systeme_monnaie) - 1

    while somme_a_rendre > 0:
        valeur = systeme_monnaie[i]
        if somme_a_rendre < valeur:
            i -= 1
        else:
            lst_pieces.append(valeur)
            somme_a_rendre -= valeur
    return lst_pieces
```

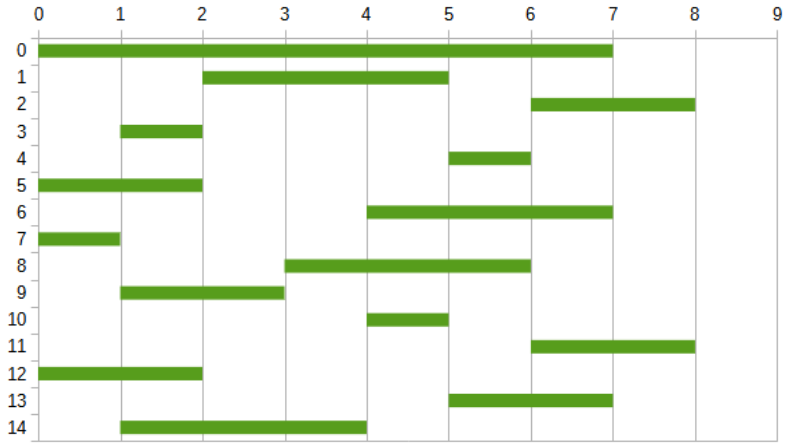
Q12. (si on a le temps)
Un autre exemple classique, sans doute plus délicat, est celui de l'allocation de ressources (salle, matériel, etc.). On suppose qu'on a n demandes de réservation pour cette ressources, avec pour chacune une heure de début et une heure de fin (et on suppose qu'on a pas besoin de période de battement). On suppose bien entendu que la

1. `https://fr.wikipedia.org/wiki/Livre_sterling` section "5.1 Avant la décimalisation"

ressource ne peut être réservée que par une personne à la foi à un instant donné. L'objectif est de satisfaire un maximum de personnes.
Formalisons le problème :
On a n le nombre de personnes, pour tout $k \in [\![1,n]\!]$ on note (d_k, f_k) le début et la fin de la demande de réservation de la k-ième personne.
On cherche une partie $J \subset [\![1,n]\!]$ (les personnes qu'on choisit pour la réservation) telle que les réservations soient compatible (ie si $(i,j) \in J^2$ alors soit $d_i \geq f_j$ soit $d_j \geq f_i$) et tel que le nombre d'élément dans J soit maximale.

L'idée ici est de considérer l'instant 0 et de choisir la personne dont la fin de la réservation se termine en premier (histoire de laisser plus de place pour les autres), disons que c'est i, maintenant l'instant f_i joue le rôle de l'instant initial, on met alors de côté toutes les personnes dont le début de réservation doit être strictement avant, parmi les personnes restante on choisit celle dont la fin de réservation se termine en premier, disons que c'est j, et on recommence avec f_j comme instant initial et on recommence jusqu'à ne trouver plus personne.

Voici un exemple de demande de réservation (chaque ligne correspond à une demande).



(a) Quelle sera la solution proposée par l'idée donnée ?

(b) On va maintenant stocker les données en mémoire, une première idée serait pour cela de poser `d=[0,2,6,1,5,0,4,0,3,1,4,6,0,5,1]` et `f=[7,5,8,2,6,2,7,1,6,3,5,8,2,7,4]`, ainsi `d[k]` (resp. `f[k]`) correspond au début d_k (resp. à la fin f_k) de la demande de la k-ième personne. Cependant on va être embêté quand on voudra trier par ordre croissant des fins, pour cela on va plutôt utiliser une liste dont les éléments sont des listes de la forme $[d_k, f_k, k]$, cela revient donc à poser :
`lst_inter=[[0,7,0], [2,5,1], [6,8,2], [1,2,3], [5,6,4], [0,2,5], [4,7,6], [0,1,7], [3,6,8], [1,3,9], [4,5,10], [6,8,11], [0,2,12], [5,7,13], [1,4,14]]`
On utilisera la commander `sorted()` (commande non exigible) où on demandera de trier par rapport à la deuxième donnée via :
`lst_inter = sorted(lst_inter, key = lambda a:a[1])`
Ainsi on a : `print(lst_inter)`
`[[0, 1, 7], [1, 2, 3], [0, 2, 5], [0, 2, 12], [1, 3, 9], [1, 4, 14], [2, 5, 1], [4, 5, 10], [5, 6, 4], [3, 6, 8], [0, 7, 0], [4, 7, 6], [5, 7, 13], [6, 8, 2], [6, 8, 11]]`
On a maintenant tout pour terminer l'algorithme. Pour cela créer un planning qu'on initialisera en `tab_planning=[lst_inter[0]]`, puis de garder en mémoire le dernier qui a réserver via `dernier=0`, il ne reste plus qu'à parcourir les indices de `lst_inter` et de rajouter au fur et à mesure les réservations (on oubliera pas de vérifier que la date de début est postérieur à la date de fin du dernier)

Correction :
La solution est donnée par l'idée est : 7, 3, 1, 4, 2

Code Python :

```
lst_inter=[[0,7,0],[2,5,1],[6,8,2],[1,2,3],[5,6,4],[0,2,5],[4,7,6],[0,1,7],[3,6,8],[1,3,9],[4,5,10],

# Nombre de demande
nb_inter = len(lst_inter)

# Tri des demandes par valeurs croissantes de la fin
lst_inter = sorted(lst_inter, key = lambda a:a[1])
```

```python
    # Tableau du planning
    tab_planning=[lst_inter[0]]

    # Dernière demande satisfaite
    dernier=0

    # On parcour toutes les demandes et on rajoute au fur
    # et à mesure
    for i in range(1,nb_inter):
        # On teste si le début de la demande i est possible
        if lst_inter[i][0]>=lst_inter[dernier][1]:
            tab_planning.append(lst_inter[i])
            dernier=i
    print(tab_planning)
```