

Corrigé du TP Informatique 01

Exercice 1

1. On saisit :

```
def voisins1(L):  
    n=len(L)  
    dmin=abs(L[0]-L[1])  
    for i in range(1,len(L)):  
        for j in range(i):  
            dcur=abs(L[i]-L[j])  
            if dcur<dmin:  
                dmin=dcur  
    return dmin
```

2. On a deux boucles **for** imbriquées d'où un coût en

$$\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} O(1) = \sum_{i=1}^{n-1} O(i) = O(n^2)$$

3. On saisit :

```
def voisins2(L):  
    tab=sorted(L)  
    dmin=tab[1]-tab[0]  
    for k in range(1,len(L)-1):  
        dcur=tab[k+1]-tab[k]  
        if dcur<dmin:  
            dmin=dcur  
    return dmin
```

4. Sans conteste après expérimentation, la version **voisins2** est préférable. La complexité temporelle de **voisins2** est donnée par le coût de **sorted** plus le nombre de passages dans la boucle **for** ce qui fait

$$O(n \log(n)) + O(n) = O(n \log(n))$$

Ainsi

La fonction **voisins2** admet une complexité temporelle en $O(n \log(n))$.

Exercice 2

1. On saisit :

```
def seq_cons(T):  
    n=len(T)  
    res=[]  
    for k in range(n-1): # on parcourt T  
        if T[k]+1==T[k+1]: # si deux entiers consécutifs  
            res.append(k) # on conserve la position du premier  
    return res
```

2. On saisit :

```
def seq_prem(T):
    n=len(T)
    k=0                                # k compteur de position
    while k<n-1 and T[k]+1!=T[k+1]: # tant que entiers non consécutifs
                                    # et avant dernier de T non atteint
        k+=1                        # on incrémente k
    if k<n-1:                        # si entiers consécutifs
        return k                    # renvoie position
    else:
        return False                # sinon renvoie False
```

Exercice 3

1. On saisit :

```
def compte(L,elt):
    res=0
    for x in L:
        res+=x==elt
    return res
```

2. On saisit :

```
def freqmax1(L):
    freq=[compte(L,x) for x in L]
    ind=0
    for k in range(len(freq)):
        if freq[k]>freq[ind]:
            ind=k
    return L[ind]
```

3. La construction de la liste `freq` est de complexité temporelle en $\sum_{i=1}^n O(n) = O(n^2)$. La boucle `for k` est simplement de complexité temporelle $O(n)$. Ainsi

La complexité temporelle de `freqmax1` est en $O(n^2)$.

Remarque : On pourrait, en restant fidèle à cette approche, améliorer un peu la complexité :

```
def compte_deb(L,deb):
    res=0
    for k in range(deb,len(L)):
        if L[k]==L[deb]:
            res+=1
    return res

def freqmax(L):
    freq=[compte_deb(L,k) for k in range(len(L))]
    ind=0
    for k in range(len(freq)):
        if freq[k]>freq[ind]:
            ind=k
    return L[ind]
```

La démarche consiste à compter les occurrences d'un élément sur les indices qui suivent cet élément puis à choisir le maximum de ce comptage. On peut vérifier formellement que le résultat est conforme et la construction de la liste `freq` requiert de l'ordre de $n^2/2$ opérations contre n^2 dans la première version. C'est donc un peu mieux sans que ce soit spectaculaire.

4. On saisit :

```
def freqmax2(L):
    tab=sorted(L)
    oc,ic=1,0    # occurrence, indice élément courant
    om,im=1,0    # occurrence, indice élément plus fréquent
    for k in range(len(tab)):
        if k==len(tab)-1 or tab[k]!=tab[k+1]: # si fin de liste ou changement
            if oc>om:                            # élément courant plus fréquent?
                om,im=oc,ic
            if k<len(tab):                        # si changement
                oc,ic=1,k+1                        # alors nouvel élément courant
        else:                                     # sinon (ni fin de liste, ni changement)
                                                    # incrémente nb d'occurrences élément courant
            oc+=1
    return tab[im]
```

La deuxième version est clairement préférable. La complexité temporelle de `sorted(L)` est en $O(n \log(n))$. Dans la boucle `for`, il n'y a que des instructions en $O(1)$. Par conséquent, la complexité temporelle est en

$$O(n \log(n)) + O(n) = O(n \log(n))$$

Ainsi

La complexité temporelle de `freqmax2` est en $O(n \log(n))$.

5. La meilleure approche consiste à utiliser des dictionnaires. On saisit :

```
def freqmax3(L):
    dico={}
    for x in L:
        if x in dico:
            dico[x]+=1
        else:
            dico[x]=1
    occ,val=0,0
    for x in dico:
        if dico[x]>occ:
            occ,val=dico[x],x
    return val
```

Le test d'appartenance, d'écriture ou de modification d'une valeur dans un dictionnaire est en $O(1)$. La première boucle est donc en $O(n)$ et le dictionnaire `dico` contient au plus n clés d'où une complexité en $O(n)$ pour la deuxième boucle. On conclut

La complexité temporelle de `freqmax3` est en $O(n)$.

Exercice 4

1. On saisit :

```
def fragmente1(L):
    n=len(L);res=[]
    djvu=[False]*n
    for k in range(n):
        if not djvu[k]:
            djvu[k]=True
            aux=[k]
            for i in range(k+1,n):
                if L[i]==L[k]:
                    djvu[i]=True
                    aux.append(i)
            res.append(aux)
        else:
            aux=[]
    return res
```

2. La construction de `djvu` est en $O(n)$ puis on a deux boucles `for` imbriquées et le reste des instructions est en $O(1)$. Si tous les éléments sont distincts, la deuxième boucle sera réalisée à chaque passage dans la première boucle ce qui fait un coût en

$$\sum_{k=0}^{n-1} \sum_{i=k+1}^{n-1} O(1) = O(n^2)$$

Si au contraire tous les éléments sont égaux, on rentre une unique fois dans la deuxième boucle ce qui fait un nombre de passage dans les boucles en $O(n)$. Ainsi

La complexité temporelle de `fragmente` est en $O(n^2)$ dans le pire des cas et en $O(n)$ dans le meilleur des cas.

Remarque : On peut omettre l'utilisation d'une liste de booléens :

```
def fragmente1(L):
    res,elt=[],[]
    n=len(L)
    for k in range(n):
        if L[k] not in elt:
            elt.append(L[k])
            aux=[k]
            for i in range(k+1,n):
                if L[i]==L[k]:
                    aux.append(i)
            res.append(aux)
    return res
```

Cette version est plus légère à écrire mais l'opération `L[k] not in elt` est tout sauf élémentaire.

2. La meilleure approche consiste à utiliser des dictionnaires. On saisit :

```
def fragmente2(L):
    dico={}
    n=len(L)
    for k in range(n):
        if L[k] in dico:
            dico[L[k]].append(k)
        else:
            dico[L[k]]=[]
    return list(dico.values())
```

Le test d'appartenance, de création d'une nouvelle entrée ou modification d'une entrée avec augmentation de la taille sont en $O(1)$. La boucle est donc en $O(n)$ et comme il y a au plus n valeurs dans `dico`, la dernière conversion en `list` est en $O(n)$ et on conclut

La complexité temporelle de `fragmente2` est en $O(n)$.

Exercice 5

1. On saisit :

```
def rech_mot(mot, texte, ind_texte):
    fini=False
    ind_mot=len(mot)
    coincide=True
    cpt=0
    while not fini:
        ind_mot-=1
        fini=ind_mot==0
        if mot[ind_mot]==texte[ind_texte]:
            if coincide:
                ind_texte-=1
            else:
                fini=True
        else:
            coincide=False
            cpt+=1
    return coincide, cpt
```

La démarche dans la boucle `while` est la suivante : s'il y a coïncidence des caractères, soit le booléen `coincide` vaut `True` et on avance potentiellement dans la détection du mot, soit le booléen `coincide` vaut `False` donc on sait que le mot est absent mais le caractère courant du texte est présent dans le mot et on peut sortir en renvoyant le décalage correspondant. À chaque désaccord, le compteur `cpt` pour le décalage est incrémenté de 1 et le booléen est basculé à `False` (cette affectation est clairement redondante). Dans tous les cas, si le début du mot est atteint, on provoque la sortie.

2. On saisit :

```
def rech(mot,texte):
    n=len(texte)
    m=len(mot)
    fini=False
    ind_texte=m-1
    while not fini:
        flag,shift=rech_mot(mot,texte,ind_texte)
        ind_texte+=shift
        fini=flag or ind_texte>=n
    return flag
```