Chapitre 5 : Travail sur les listes

I) Rappels sur les listes

- Une liste est un ensemble ordonné de valeurs (nommées éléments de la liste), repérées par leur index dans la liste.

- Pour construire une liste, on énumère les éléments de la liste entre crochets, séparés par des virgules :

- Par exemple : L=[3,7,42,1,4,8,12]

- Il est également possible de créer une liste par concaténation de 2 listes :

```
c=[0,1]+[2,4]     #résultat : c=[0,1,2,4]
d=3*[0,1]         #résultat : d=[0,1,0,1,0,1]
```

- Les opérations « + » et « * » ne réalisent donc pas l'addition ou la multiplication élément par élément.

- Les éléments d'une liste sont numérotés ou indicés à partir de 0 : « L[2] » correspond donc au 3e élément de la liste L.

- Pour modifier la deuxième case de a et lui donner la valeur « 0 », il suffit d'écrire : « L[1] = 0 ».

- La fonction « len(L) » permet de connaître le nombre d'éléments de L.

- On peut aussi facilement accéder aux derniers éléments d'une liste : « L[len(L)] » ou « L[-1] » correspond au dernier élément de L par exemple.

- On peut également extraire tous les éléments situés entre les indices i (inclus) et j (exclus) d'une liste a avec la notation « a[i:j] ». Cette notation crée donc une liste de longueur j-i.

```
a=[3,7,42,1,4,8,12]
b=a[2:5]              #résultat : b = [42,1,4]
```

Pour ce chapitre, nous travaillerons exclusivement avec des listes de nombres réels. Nous pourrons ainsi faire des calculs sur ces valeurs.

II) Les premiers algorithmes sur les listes

1) Calcul de la moyenne des éléments d'une liste.

Les deux fonctions suivantes (ayant pour entrée n) renvoient une liste de n valeurs entières au hasard entre 0 et 20. La première utilise une liste L définie en compréhension, mais ces deux algorithmes sont aussi efficaces l'un que l'autre.

```
def listealea(n):
    L=[randint(0,20) for k in range(0,n)]
    return(L)
```

```
def listealea2(n):
    L=[]
    for k in range(0,n):
        L=L+[randint(0,20)]
    return(L)
```

Exemple : écrire une fonction qui prend en entrée une liste de nombres réels et qui renvoie la moyenne de ces valeurs.

Algorithme en langage naturel :

Entrée : L (liste)
Traitement :
S ← 0
$n \leftarrow$ len(L)
« c'est le nombre d'éléments de la liste L »
Pour k dans L
    $S + k \leftarrow S$
Fin pour
« S contient la somme des éléments de L en sortie de boucle »
Sortie : Retourner(S/n)

Code Python :

```
14  def moyenne(L):
15      S=0
16      n=len(L)
17      for k in L:
18          S=S+k
19      return(S/n)
20
21  liste1=listealea(10)
22  print(liste1)
23  print(moyenne(liste1))
```

Prop : Cet algorithme a une complexité temporelle en $O(n)$ où n correspond au nombre d'éléments de la liste L. En effet on réalise 2 opérations (une affectation et une somme) à chaque passage dans la boucle et on passe n fois dans la boucle. On ajoute 3 opérations (2 affectations et une division) pour avoir donc en tout $2n + 3$ opérations élémentaires.

2) Détermination du maximum d'une liste.

Ecrire une fonction nommée maximum qui prend en entrée une liste L et qui renvoie la plus grande valeur de L.

Algorithme en langage naturel :

Entrée : L (liste)
Traitement :
M ← $L[0]$
Pour k dans L
    Si $k > M$ alors
        $M = k$
    Fin si
Fin pour
« M contient le plus grand élément de L en sortie de boucle »
Sortie : Retourner(M)

Code Python :

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

Prop : Cet algorithme a une complexité temporelle en $O(n)$ où n correspond au nombre d'éléments de la liste L. En effet on réalise au maximum 2 opérations (une affectation et une comparaison) à chaque passage dans la boucle et on passe n fois dans la boucle. On ajoute 1 opération (M=L[0]) pour avoir donc en tout $2n + 1$ opérations élémentaires dans le pire des cas.

Exemple : Ecrire un algorithme qui détermine les deux plus grandes valeurs (différentes) d'une liste L. Attention, il faut prendre en compte que le maximum peut apparaitre plusieurs fois.

Une idée est de trouver le maximum M de la liste L (avec la fonction précédente) puis de construire une nouvelle liste L2 en y mettant tous les éléments de L différents de M. Puis on appliquera la fonction maximum à la liste L2 pour avoir la 2e plus grande valeur (différente) de L

Algorithme en langage naturel :

Entrée : L (liste)
Traitement :
M_1 ←maximum(L)
« M_1 contient le plus grand élément de L »
$L_2 \leftarrow [\,]$
Pour k dans L
    Si $k! = M_1$ alors
        $L_2 \leftarrow L_2 + [k]$
    Fin si
Fin pour
« en sortie de boucle L_2 contient tous les éléments de L sauf le maximum »
M_2 ←maximum(L_2)
« M_2 contient le plus grand élément de L_2 »
Sortie : Retourner(M_1, M_2)

Code Python :

```
def maximum2(L):
    M1=maximum(L)
    L2=[]
    for k in L:
        if k!=M1:
            L2=L2+[k]
    M2=maximum(L2)
    return(M1,M2)
```

III) Recherche dans une liste

1) Recherche séquentielle d'un élément

L'objectif est ici de déterminer si une liste L prédéfinie contient une certaine valeur x.

La fonction suivante retourne la valeur « True » si x est élément de L, et « False » sinon.

Algorithme en langage naturel :

Entrée : L (liste) ; x (float)
Traitement :
Pour k dans L
  Si k==x alors
    Renvoyer (true)
    Stop (break)
Renvoyer (false)

Code Python :

```
def estdans(L,x):
    for k in L:
        if k==x:
            return (True)
            break
    return (False)
```

Rq : Attention aux majuscules dans « True » et « False »

Prop : Cet algorithme a une complexité en $O(n)$ où n est le nombre d'éléments de L puisqu'au pire des cas (si x n'est pas dans L) on effectue une comparaison à chaque passage dans la boucle, et on fait n passages dans la boucle.

2) Recherche par dichotomie dans une liste triée

On suppose à présent que l'on traite une liste triée, c'est-à-dire composée de réels croissants.

La recherche d'un élément x peut ainsi être rendue plus efficace que la recherche séquentielle.

L'idée de la dichotomie est la suivante : on coupe la liste en deux et on recherche si la valeur x doit être recherchée dans la moitié gauche (inférieure) ou droite (supérieure). Il suffit pour cela de comparer x avec la valeur centrale de la liste, puis de réduire la recherche à la moitié de la liste, et ainsi de suite.

Voici le principe sur un exemple simple :

Données : $L = [1,3,5,6,9,12,14]$; élément recherché $x = 9$.

Nombre d'éléments de L : $len(L) = 7$. L'élément central est donc $L[3] = 6$

On compare x avec $L[3] = 6 : x > L[3]$ donc il faut chercher x dans $L[4:6]$

Nombre d'éléments de $L[4:6] : 3$. On compare x avec $L[5] = 12 : x < L[5]$, donc il faut chercher dans $L[4:4]$.

Nombre d'éléments de $L[4:4] : 1$. Il suffit alors de regarder si cet élément est égal à x ou pas.

Rq : Dans ce cas, l'algorithme ne réalisera que 3 tests au maximum. A noter qu'il faut prévoir dans le code la possibilité de tomber sur la bonne valeur.

```
def dicho(L,x):
    while(len(L)>1):
        rangmilieu=floor(len(L)/2) #c'est l'indice du terme du milieu de L
        if x==L[rangmilieu]:# si le terme du milieu est x c'est gagné
            return(True)
            break
#Si x est plus grand, on poursuit dans la partie droite de L
        elif x>L[rangmilieu]:
            L=L[rangmilieu:]
        else:
#Si x est plus petit, on poursuit dans la partie gauche de L
            L=L[:rangmilieu]
# en sortie de boucle, il ne reste qu'un élément dans L, on regarde si c'est x
    if L[0]==x:
        return(True)
    else:
        return(False)

print(dicho([1,3,5,8,9,45,74,86,122],8))
```

Rq : Notons que cet algorithme se termine toujours puisque la taille de L est divisée par 2 à chaque passage dans la boucle. Ainsi, il existera bien un moment pour lequel la taille de L sera ≤ 1.
Notons également, que même si la liste L est de très grande taille, cet algorithme va très vite puisqu'après k passages dans la boucle while, la taille de L a été divisée par 2^k.

Prop : Soit n le nombre d'éléments de la liste L (ou sa taille si vous préférez). La complexité de cet algorithme est en $O(\ln(n))$. Il s'agit donc d'une complexité logarithmique, et cet algorithme est donc extrêmement performant.

Preuve : A chaque passage dans la boucle « while », on divise par 2 la taille de L. Si on note p le nombre de passages dans la boucle, alors p est le plus petit entier tel que $\frac{n}{2^p} < 1$, soit $n < 2^p$ et donc $\ln(n) < p\ln(2)$ donc $p = E\left(\frac{\ln(n)}{\ln(2)}\right) + 1$ où $E(x)$ désigne la partie entière de x.
De plus à chaque passage dans la boucle, on réalise un nombre constant d'opération (dans le pire des cas : 5 opérations) donc la complexité est en $O\left(5\left(\frac{\ln(n)}{\ln(2)} + 1\right)\right) = O(\ln(n))$

Rq : Cet algorithme est donc plus rapide que la recherche séquentielle, mais ne peut s'appliquer qu'à une liste triée. Trier une liste demande un algorithme plus ou moins complexe pour se faire.

IV) Algorithmes de tris sur une liste

Le tri par ordre croissant (ou décroissant) (en anglais *sort*) d'une liste est l'une des premières problématiques de l'algorithmique moderne et a fait naître de nombreuses familles d'algorithmes, ainsi que des recherches mathématiques dédiées. Il s'agit d'un outil omniprésent en informatique, que ce soit dans les sites de référencement de produits, dans les tableurs, etc…

Les listes de données étant toujours plus grandes, les algorithmes de tri doivent être efficaces. Pour une liste de longueur n, les meilleurs algorithmes de tri ont une complexité moyenne en $n.ln(n)$, ou quasi-linéaire.

Les algorithmes de tri de complexité quadratique sont les plus simples à imaginer.

Le tri par sélection :

L'idée est de chercher dans la liste le plus petit élément (en parcourant donc toute la liste), et de le placer en 1er élément par un échange. Ensuite, on cherche, à partir du 2e élément de cette nouvelle liste, le plus petit élément, qu'on place en 2e, et ainsi de suite.

Def : En Python, on peut affecter plusieurs valeurs à plusieurs variables en même temps, ce qui nous dispense de créer des variables de stockage temporaires. On utilise la syntaxe suivante :

« variable1 , variable2, …, variable n = valeur1 , valeur 2 , … valeur n »

```
1  from random import*
2  from math import*
3  def echange(i,j,L):
4  #permet d'échanger les éléments d'indices i et d'indice j de la liste L
5      L[i],L[j]=L[j],L[i]
6      return(L)
7
8  def minimum(L):
9  # renvoie la valeur minimale m d'une liste L et son rang r
10     m=L[0]
11     r=0
12     for k in range(0,len(L)):
13         if L[k]<m:
14             m=L[k]
15             r=k
16     return(m,r)
17
18 def triselection(L):
19     for j in range(0,len(L)):
20 #on prend le rang du plus petit élément de L après le rang j
21 #attention le rang d'un élément de L[j:] est décalé de j par rapport
22 #au rang du même élément dans L
23         minj=minimum(L[j:])[1]
24 # on le met à la place de celui de rang j
25         L=echange(j+minj,j,L)
26     return(L)
27
28 liste=[1,20,3,41,5,67,7,10]
29 print(liste)
30 print(triselection(liste))
```

Prop : La complexité de la fonction « echange » est constante (2 affectations)
Celle de la fonction « minimum » est en $O(n)$ où n est la taille de L (comme pour la fonction maximum).
Enfin dans la fonction « triselection » on passe n fois dans la boucle « pour » et à chaque passage on fait appel à la fonction « minimum » et à la fonction « echange ».
On réalise donc de l'ordre de n^2 opérations élémentaires, d'où une complexité en $O(n^2)$

Le tri par insertion :

L'idée est de partir d'une liste vide M et d'insérer un à un les éléments de la liste L à trier en les comparant aux éléments de M pour les insérer au bon endroit de sorte que M soit triée par ordre croissant.

```
1  from random import*
2  from math import*
3  def position(x,L):
4  #retourne la position à laquelle insérer x dans la liste L triée par ordre croissant
5      for rang in range(0,len(L)):
6          if L[rang]>x:
7              return(rang)
8              break
9  # si on ne passe jamais dans la boucle conditionnelle, c'est que x est supérieur
10 # à toutes les valeurs de L, on l'insère donc à la fin
11     return(len(L))
12
13 def triinsertion(L):
14     M=[]
15     for x in L:
16 #pour chaque élément de L, on utilise la fonction précédente pour savoir
17 # à quel rang l'insérer dans M, et on l'insère à ce rang
18         r=position(x,M)
19         M.insert(r,x)
20     return(M)
21 liste=[10,54,87,54,21,65,9,125]
22 print(triinsertion(liste))
```

Prop : La complexité de la fonction « position » est en $O(n)$, où n est la taille de L (on passe au maximum n fois dans la boucle et à chaque passage on fait 1 comparaison).
Enfin dans la fonction « triinsertion » on passe n fois dans la boucle « pour » et à chaque passage on fait appel à la fonction « position »
On réalise donc de l'ordre de n^2 opérations élémentaires, d'où une complexité en $O(n^2)$

Le tri à bulles (ou par propagation) ou le tri par insertion sont également quadratiques.

Tri à bulles :

Le tri à bulles est un algorithme simple à comprendre :

- on parcourt toute la liste en permutant 2 à 2 les éléments s'ils ne sont pas dans le bon ordre : l'élément le plus grand « remonte » ainsi en dernière position,

- on reproduit l'opération, en se limitant aux $n-1$ premiers éléments, potentiellement encore en désordre.

La complexité de cet algorithme est quadratique : on réalise n tests (et permutations), puis $n-1$, etc, soit un nombre d'opérations de l'ordre de n^2 (dans le pire des cas).
On rappelle en effet que $1+2+3+\cdots+n-1+n=\frac{n(n+1)}{2}=\frac{n^2+n}{2}$

```python
def tribulle(L):
    for i in range(0,len(L)):
        for j in range(0,len(L)-1-i):
            if L[j]>L[j+1]:
                L[j+1],L[j]=L[j],L[j+1]
    return(L)
```

rq : Il existe de nombreux algorithmes de tri en complexité quasi-linéaire (et donc bien meilleurs que des tris de complexité quadratique). Chacun possède sa propre logique. Les plus utilisés sont le tri rapide (quick sort), le tri par fusion (merge sort) ou le tri de Shell (shell sort). Mais beaucoup reposent sur le principe de récursivité que nous verrons dans un chapitre ultérieur.

Exercices :

1) Ecrire un algorithme ayant pour entrée une liste L qui renvoie le plus petit élément de L.

2) Ecrire un algorithme ayant comme entrée une liste L et un élément x et qui renvoie le nombre de fois que l'élément x est présent dans la liste L.

3) Ecrire un programme réalisant un tri par insertion à l'aide de deux boucles imbriquées (sans fonction préalable nécessaire donc)

Correction question 3

```
def triinsertion2(L):
    M=[]
    for k in range(0,len(L)):
        insertion=False
# ceci pertra de savoir si on a inséré L[k] dans M ou pas.
# si ce n'est pas le cas, c'est qu'il est plus grand que tous les éléments de M
# il faudra donc l'insérer à la fin
        for t in range(0,len(M)):
            if L[k]<M[t]:
                M.insert(t,L[k])
                insertion=True
        if insertion==False:
            M.insert(len(M),L[k])

    return(M)
print(triinsertion2([1,54,2,36,47]))
```