Exercice 1 : Tri par sélection

Réaliser un algorithme itératif de tri par sélection en utilisant deux fonctions :

- Une fonction minimum (L,k) qui renvoie l'index de la position du minimum d'une liste L étudiée à partir de l'index k.
- Une fonction $tri(L)$ qui retourne la liste L triée en utilisant la fonction *minimum* précédente

```
def minimum(L,k):
    index_min=k
    for j in range(k+1,len(L)):
        if L[j]<L[index_min]:
                index_min=j
    return index_min
def tri2(L):
    for i in range((len(L)-1)):
        index_min=minimum(L,i)
        L[i],L[index_min]=L[index_min],L[i]
    return L
```

Exercice 2 : Tri par sélection récursif

On rappelle l'algorithme itératif du cours du tri par sélection :

```
def selection(L):
    for i in range(len(L)-1):
        mini=L[i]
        indice=i
        for j in range(i+1,len(L)):
            if L[j]<mini:
                mini=L[j]
                indice=j
        L[i],L[indice]=L[indice],L[i]
    return L
```

Proposer une réécriture du code précédent afin d'effectuer un tri par sélection récursif

```
def selection_rec(L):
    """le pb de cette méthode est qu'elle
    est de complexité spatiale quadratique"""
    if len(L)==0:
        return []
    else:
        mini=L[0]
        indice=0
        for j in range(1,len(L)):
            if L[j]<mini:
                mini=L[j]
                indice=j
        L[0],L[indice]=L[indice],L[0]
        return [L[0]]+selection_rec(L[1:])
```

Exercice 3 : Tri par insertion récursif

On rappelle la version itérative du cours de ce tri :

```
def tri_insertion(L):
    for i in range(1,len(L)):
        cle=L[i]
        j=i-1
        while j>=0 and L[j]>cle:
            j=j-1
        L[j+2:i+1]=L[j+1:i]
        L[j+1]=cle
    return L
```

Ecrire une fonction tri_rec( ) récursive qui :

- Prend pour argument une liste L de longueur n ainsi qu'un entier i (qui va jouer le même rôle que dans la version itérative). Au premier appel, on fixe i=1
- Le cas trivial stoppant le tri est celui pour lequel i=n
- Dans les autres cas, cette fonction tri par insertion la liste jusqu'à l'index i et effectue un appel récursif pour continuer le tri.

```
def insertion_rec(L,i):
    if i==len(L)-1:
        return L
    else :
        cle=L[i]
        j=i-1
        while j>=0 and L[j]>cle:
            j=j-1
        L[j+2:i+1]=L[j+1:i]
        L[j+1]=cle
        return insertion_rec(L,i+1)
```

Exercice 4 : Tri par insertion décroissant

Ecrire un programme itératif de tri par insertion décroissant en utilisant la méthode *enumerate*().

```
def tri_enumerate(L):
    for i,cle in enumerate(L):#obligé de tout prendre !
        j=i-1
        while cle>L[j] and j>=0:
            j=j-1
        L[j+2:i+1]=L[j+1:i]
        L[j+1]=cle
    return L
```

Exercice 5 : Tri par insertion (2e version)

Nous avons vu dans le cours le tri par insertion en comparant la clé $L[i]$ en partant de la fin de la sous-listes déjà triée. On peut envisager le tri par insertion itératif en comparant la clé en partant du début de la sous-liste déjà triée. Proposer un programme permettant de réaliser ce type de tri.

```
def tri_insertion(L):
    for i in range(1,len(L)):
        cle=L[i]
        j=0
        while j<i and cle>L[j]:
            j=j+1
        L[j+1:i+1]=L[j:i]
        L[j]=cle
    return L
```

- Meilleur cas : $T(n) = O(n)$
- Pire cas : $T(n) = O(n^2)$

Exercice 6 : Temps d'exécution

Pour mesurer le temps d'exécution d'un programme, on importe la fonction *time* du module *time* avec l'instruction :

```
from time import time
```

Pour générer un entier tiré au hasard, on importe la fonction $np.random.randint(debut, fin)$ du module *numpy*

On souhaite comparer les temps d'exécution du tri insertion et du tri fusion sur deux types de listes : une liste de nombres pris au hasard et une liste de nombre déjà triée.

1) Construire une fonction *liste_alea* de la variable n générant une liste de n entiers pris au hasard entre 1 et 10000, bornes comprises.
2) Construire une fonction *liste_triee* de la variable n générant une liste de n entiers triés de manière croissantes ente 0 à $n-1$, bornes comprises.
3) Mesurer les temps d'exécution des programmes de tri insertion et de tri fusion pour trier des listes déjà triées puis des listes de nombres aléatoires pour $n = 20000$ et $n = 40000$. Conclure.

```
def liste_aleatoire(n):
    L=[]
    for i in range(n):
        L.append(np.random.randint(1,10000+1))
    return L

def liste_triée(n):
    L=[]
    for i in range(n):
        L.append(i)
    return L


def temps(liste_n):
    temps_trie_fusion=[]
    temps_pas_trie_fusion=[]
    temps_trie_insertion=[]
    temps_pas_trie_insertion=[]
    for i in liste_n:
        L=liste_triée(i)
        t0=time()
        fusion2(L)
        temps_trie_fusion.append(time()-t0)
        L=liste_aleatoire(i)
        t0=time()
        fusion2(L)
        temps_pas_trie_fusion.append(time()-t0)
        L=liste_triée(i)
        t0=time()
        insertion(L)
        temps_trie_insertion.append(time()-t0)
        L=liste_aleatoire(i)
        t0=time()
        insertion(L)
        temps_pas_trie_insertion.append(time()-t0)
    return print("liste de n ={0}\n\
temps de tri fusion pour une liste triée ={1}, \n\
temps de tri fusion pour une liste aléatoire ={2}\n\
temps de tri insertion pour une liste triée = {3} \n\
temps de tri insertion pour une liste aléatoire =
{4}".format(liste_n,temps_trie_fusion,

temps_pas_trie_fusion,temps_trie_insertion,temps_pas_trie_ins
ertion))
```

```
liste de n =[20000, 40000]

temps de tri fusion pour une liste triée =[0.08577346801757812, 0.15860986709594727],

temps de tri fusion pour une liste aléatoire =[0.10372138023376465, 0.21943378448486328]

temps de tri insertion pour une liste triée = [0.008975982666015625, 0.019886016845703125]

temps de tri insertion pour une liste aléatoire = [12.517660140991211, 53.17120027542114]
```

On retrouve :

- Une complexité en $nlog_2(n)$ donc quasiliénaire dans tous les cas pour le tri fusion
- Une complexité linéaire si la liste est triée pour le tri insertion
- Une complexité « gourmande » si la liste est aléatoire et triée avec un tri insertion

Exercice 7 : Ordre lexicographique

L'objectif est d'écrire un programme qui trie une liste de mots (type *str*) suivant l'ordre lexicographique.

1) Ecrire une fonction *ordre_alphabetique* qui prend en argument deux caractères alphabétiques $c1$ et $c2$ et renvoie -1 si $c1$ est avant $c2$, 1 si $c2$ est avant $c1$ et 0 si $c1 = c2$.
2) Ecrire une fonction *ordre_lexicographique* qui prend pour argument deux mots différents $m1$ et $m2$ et renvoie -1 si "m1" < "$m2$" pour l'ordre lexicographique, 0 si "$m1 = m2$" et 1 si "$m1 > m2$". On utilisera la fonction *ordre_alphabetique*
3) Ecrire une fonction *tri_lexicographique* qui prend en argument une liste de mots et trie cette liste. On utilisera la fonction *ordre_lexicographique* avec l'algorithme de tri par insertion.

```
def ordre_alphabetique(c1,c2):
    if c1<c2:
        return -1
    elif c1>c2:
        return 1
    else :
        return 0
def ordre_lexicographique(m1,m2):
    N=min([len(m1),len(m2)])
    for i in range(N):
        if ordre_alphabetique(m1[i],m2[i]) ==-1:
            return -1
        elif ordre_alphabetique(m1[i],m2[i]) ==1:
            return 1
    if len(m2)>N:#un cas suffisant pour traiter les racines
communes
        return -1
def tri_lexicograpjique(L):
    for i in range(1,len(L)):
        cle=L[i]
        j=i-1
        while j>=0 and ordre_lexicographique(cle,L[j])==-1:
            j=j-1
        L[j+2:i+1]=L[j+1:i]
        L[j+1]=cle
    return L
```

Exercice 8 : Permutation

En s'inspirant de l'algorithme du tri sélection écrire une fonction permute qui prend en argument une liste de mots et modifie en place cette liste qui sera triée par ordre croissant de la longueur de chaque mot. La fonction ne renvoie rien.

Tester avec la liste : `L=["toto","bonjour","a","oui","non"]`

```
def permute(L):
    index_min=0
    for i in range(1,len(L)):
        if len(L[i])<len(L[index_min]):
            index_min=i
    L[0],L[index_min]=L[index_min],L[0]
```

Exercice 9 : Trier des points

On dispose de points dans le plan muni d'un repère orthonormé d'origine O. Ces points possèdent un couple de coordonnées $(x; y)$ représenté par la liste $[x, y]$. Nous allons trier ces points en fonction de leur distance au point O, de la plus petite à la plus grande.

1) Ecrire une fonction *distance* qui prend en paramètre une liste de deux nombres nommée *point* qui représente un point dans le plan (*point* est la liste des coordonnées d'un point P), et renvoie le carré de la distance de ce point O.
2) Ecrire une fonction *compare* qui prend en paramètres deux listes $point1$ et $point2$ représentant deux points P_1 et P_2et qui renvoie -1 si P_1 est plus proche de O que P_2, 1 si P_2 est plus proche de O que P_1 et 0 si les deux points sont équidistants de O.
3) Ecrire une fonction *tri_points* qui prend en paramètre une liste composée de listes de deux nombres représentant des points dans le plan et qui trie cette liste suivant la distance entre les points et O.
   a) Utiliser un algorithme de tri sélection
   b) Utiliser un algorithme de tri insertion

```
def distance(point):
    x=point[0]
    y=point[1]
    return x**2+y**2


def compare(point1,point2):
    d1=distance(point1)
    d2=distance(point2)
    if d1>d2:
        return 1
    elif d1<d2:
        return -1
    else :
        return 0


def tri_points1(liste_points):#en utilisant un tri sélection
    for i in range(len(liste_points)):
        minimum=liste_points[i]
        index_min=i
        for j in range(i,len(liste_points)):
            if compare(minimum,liste_points[j]) == 1:
                minimum=liste_points[j]
                index_min=j

liste_points[i],liste_points[index_min]=minimum,liste_points[
i]
    return liste_points

print(tri_points1([[2,2],[0,1],[10,10],[1,1]]))

def tri_points3(liste_points):#tri insertion
    for i in range(1,len(liste_points)):
        j=i-1
        cle=liste_points[i]
        while j>=0 and compare(cle,liste_points[j])==-1:
            j=j-1
        liste_points[j+2:i+1]=liste_points[j+1:i]
        liste_points[j+1]=cle
    return liste_points

print(tri_points3([[2,2],[0,1],[10,10],[1,1]]))
```

Exercice 10 : Tri par insertion dichotomique

Programmer un algorithme de tri insertion avec une insertion qui utilise la recherche dichotomique.

```
def dicho_it(L,x):
    """recherche dicho itérative
    fonction qui renvoie l'indice où
    placer l'élément x"""
    deb=0
    fin=len(L)-1
    while fin>=deb:
        m=(fin+deb)//2
        if L[m]==x:
            return m+1
        elif L[m]<x:
            deb=m+1
        else :
            fin=m-1
    return deb

def insertion_it(L):
    for i in range(1,len(L)):
        cle=L[i]
        j=dicho_it(L[:i],cle)
        L[j+1:i+1]=L[j:i]
        L[j]=cle
    return L

def dicho_rec(L,x):
    """recherche dicho récursive
    fonction qui renvoie l'indice où
    placer l'élément x"""
    def aux(L,x,indice):
        if len(L)==0:
            return indice
        else :
            m=len(L)//2
            if L[m]==x:
                return indice+m+1#on cherche la position pour
insérer
            elif L[m]<x:
                return aux(L[m+1:],x,indice+m+1)
            else :
                return aux(L[:m],x,indice)
    return aux(L,x,0)
```

```
def insertion_dicho(L):
    def aux1(L,i):
        if i==len(L):
            return L
        else :
            cle=L[i]
            j=dicho_rec(L[:i],cle)
            L[j+1:i+1]=L[j:i]
            L[j]=cle
            return aux1(L,i+1)
    return aux1(L,1)
```

Exercice 11 : Tri sélection amélioré

A l'exercice 2, nous avons réalisé une version récursive du tri par sélection. Cette version a pour défaut une complexité spatiale quadratique. Dans cet exercice, on cherchera à éviter cette complexité.

Réaliser un algorithme récursif de tri par sélection en utilisant deux fonctions :

- Une fonction minimum(L) qui renvoie, de manière récursive, l'index de la position du minimum d'une liste L.
- Une fonction tri(L) récursive qui retourne la liste L triée en utilisant la fonction minimum précédente.

```
def minimum1(L,k):
    """version qui nécessite un argument supplémentaire
    autre problème cette fonction renvoie le minimum et son
indice"""
    if k==len(L)-1:
        return L[k],k
    else :
        val,indice=minimum1(L,k+1)
        if L[k]>val:
            return val,indice
        else :
            return L[k],k


def minimum2(L):
```

```
    """pareil que minimum1 mais avec fonction auxiliaire
    et donc sans problème d'indice"""
    def auxiliaire(L,k):
        if k==len(L)-1:
            return L[k],k
        else :
            val,indice=auxiliaire(L,k+1)
            if L[k]>val:
                return val,indice
            else :
                return L[k],k
    mini,indice=auxiliaire(L,k=0)
    return indice


def tri1(L,k):
    """sans fonction auxiliaire"""
    if k==len(L)-1:
        return L
    else :
        indice=minimum2(L[k:])
        indice=indice+k
        L[k],L[indice]=L[indice],L[k]
        return tri1(L,k+1)


def tri2(L):
    """avec fonction auxiliaire"""
    def tri1(L,k):
        if k==len(L)-1:
            return L
        else :
            indice=minimum2(L[k:])
            indice=indice+k
            L[k],L[indice]=L[indice],L[k]
            return tri1(L,k+1)
    L=tri1(L,0)
    return L




def tri3(L):
    """fonction de complexité spatiale quadratique """
    if len(L)==0:
        return []
    else :
        indice=minimum2(L)
        L[indice],L[0]=L[0],L[indice]
        return [L[0]]+tri3(L[1:])
```