EXERCICE 4 (4 points)

Cet exercice porte sur l'algorithme de tri fusion, qui s'appuie sur la méthode dite de « diviser pour régner ».

1. a. Quel est l'ordre de grandeur du coût, en nombre de comparaisons, de l'algorithme de tri fusion pour une liste de longueur n ?

b. Citer le nom d'un autre algorithme de tri. Donner l'ordre de grandeur de son coût, en nombre de comparaisons, pour une liste de longueur n. Comparer ce coût à celui du tri fusion. Aucune justification n'est attendue.

L'algorithme de tri fusion utilise deux fonctions `moitie_gauche` et `moitie_droite` qui prennent en argument une liste `L` et renvoient respectivement :

- la sous-liste de `L` formée des éléments d'indice strictement inférieur à `len(L)//2` ;
- la sous-liste de `L` formée des éléments d'indice supérieur ou égal à `len(L)//2`.

On rappelle que la syntaxe `a//b` désigne la division entière de `a` par `b`.

Par exemple,

```
>>> L = [3, 5, 2, 7, 1, 9, 0]
>>> moitie_gauche(L)
[3, 5, 2]
>>> moitie_droite(L)
[7, 1, 9, 0]
```

```
>>> M = [4, 1, 11, 7]
>>> moitie_gauche(M)
[4, 1]
>>> moitie_droite(M)
[11, 7]
```

L'algorithme utilise aussi une fonction `fusion` qui prend en argument deux listes triées `L1` et `L2` et renvoie une liste `L` triée et composée des éléments de `L1` et `L2`.

On donne ci-dessous le code python d'une fonction récursive `tri_fusion` qui prend en argument une liste `L` et renvoie une nouvelle liste triée formée des éléments de `L`.

```
def tri_fusion(L):
  n = len(L)
  if n<=1 :
     return L
  print(L)
  mg = moitie_gauche(L)
  md = moitie_droite(L)
  L1 = tri_fusion(mg)
  L2 = tri_fusion(md)
  return fusion(L1, L2)
```

2. Donner la liste des affichages produits par l'appel suivant.

```
tri_fusion([7, 4, 2, 1, 8, 5, 6, 3])
```

On s'intéresse désormais à différentes fonctions appelées par `tri_fusion`, à savoir `moitie_droite` et `fusion`.

3. Écrire la fonction `moitie_droite`.

4. On donne ci-dessous une version incomplète de la fonction `fusion`.

```
1. def fusion(L1, L2):
2.     L = []
3.     n1 = len(L1)
4.     n2 = len(L2)
5.     i1 = 0
6.     i2 = 0
7.     while i1 < n1 or i2 < n2 :
8.         if i1 >= n1:
9.             L.append(L2[i2])
10.            i2 = i2 + 1
11.        elif i2 >= n2:
12.            L.append(L1[i1])
13.            i1 = i1 + 1
14.        else:
15.            e1 = L1[i1]
16.            e2 = L2[i2]
17.
18.
19.
20.
21.
22.
23.     return L
```

Dans cette fonction, les entiers `i1` et `i2` représentent respectivement les indices des éléments des listes `L1` et `L2` que l'on souhaite comparer :

- Si aucun des deux indices n'est valide, la boucle `while` est interrompue ;
- Si `i1` n'est plus un indice valide, on va ajouter à `L` les éléments de `L2` à partir de l'indice `i2` ;
- Si `i2` n'est plus un indice valide, on va ajouter à `L` les éléments de `L1` à partir de l'indice `i1` ;
- Sinon, le plus petit élément non encore traité est ajouté à `L` et on décale l'indice correspondant.

Écrire sur la copie les instructions manquantes des lignes 17 à 22 permettant d'insérer dans la liste `L` les éléments des listes `L1` et `L2` par ordre croissant.

Exercice 1 (4 points)

Cet exercice traite de programmation, d'algorithmique et de complexité.

On souhaite rechercher dans un tableau les k plus proches voisins d'un objet donné.

On dispose pour cela d'un tableau `t` non vide contenant des objets d'un même type et d'une fonction `distance` qui renvoie la distance entre deux objets quelconques de ce type.
Étant donné un objet `cible` du même type que ceux du tableau `t`, on cherche à déterminer les indices des k éléments du tableau `t` qui sont les plus proches de cet objet (c'est-à-dire ceux dont la distance à l'objet `cible` est la plus petite).

1. On suppose dans cette question que $k = 1$.
   La fonction `plus_proche_voisin(t, cible)` ci-dessous prend en argument le tableau `t` et l'objet `cible`. Écrire sur votre copie uniquement le bloc d'instructions manquant pour que la fonction renvoie l'indice d'un plus proche voisin de `cible`.

```
def plus_proche_voisin(t, cible) :
    dmin = distance(t[0], cible)
    idx_ppv = 0
    n = len(t)
    for idx in range(1, n) :
                                  } bloc à écrire
    return idx_ppv
```

2. On considère que le coût en temps du bloc manquant est constant. Quelle est la complexité de la fonction `plus_proche_voisin` quand $k = 1$?

Dans la suite, on suppose que $k \geq 1$.

3. Une approche naïve consiste à parcourir le tableau `t` pour trouver l'indice de l'élément le plus proche de `cible`, puis à recommencer pour trouver l'indice du deuxième élément le plus proche de `cible`, et ainsi de suite. Cela implique de parcourir k fois tout le tableau.
   Afin de réduire le nombre d'appels à la fonction `distance`, la stratégie suivante permet de ne parcourir le tableau `t` qu'une seule fois. Lors de ce parcours, on stocke dans une liste `kppv`, initialement vide, les tuples `(idx, d)` où `idx` est l'indice d'un k plus proche voisin de `cible` déjà rencontré et `d` la distance correspondante, triés dans l'ordre décroissant de leur distance à `cible`.
   La fonction `recherche_kppv(t, k, cible)` ci-après renvoie ainsi la liste des tuples `(idx, d)` où `idx` est l'indice d'un k plus proche voisin de `cible` dans le tableau `t` et `d` la distance correspondante.

On admet que la fonction `insertion(kppv, idx, d)` insère le tuple `(idx, d)` dans la liste `kppv` de sorte que celle-ci demeure triée dans l'ordre décroissant des distances.

```
def recherche_kppv(t, k, cible):
    kppv = []
    n = len(t)
    for idx in range(n):
        obj = t[idx]
        if len(kppv)<k:
            insertion(kppv, idx, distance(obj, cible))
        else:
            i0, d0 = kppv[0]
            if distance(obj, cible)<d0:
                kppv.pop(0) # supprime le 1er élément de kppv
                insertion(kppv, idx, distance(obj, cible))
    return kppv
```

a. On remarque qu'il y a plusieurs appels identiques à la fonction `distance(obj,cible)`. Comment ne faire qu'un seul appel de cette fonction ?

b. Expliquer l'intérêt de maintenir la liste `kppv` triée.

c. Écrire une fonction `insertion(kppv, idx, d)` qui insère le tuple `(idx, d)` dans la liste `kppv` préalablement triée en préservant l'ordre décroissant selon l'élément `d`.

On pourra éventuellement utiliser la méthode `insert` dont la documentation, fournie par la commande `help(list.insert)`, est la suivante :

```
insert(self, index, object, /)
 Insère l'objet object avant la position index dans l'objet
appelant référencé par self.
```

Exemple d'utilisation de la méthode `insert` :

```
>>> liste = [4, 2, 8, 9]
>>> liste.insert(1, 3)
>>> liste
[4, 3, 2, 8, 9]
```