TD7 : AP.

Réalisez un prédicat qui

- prend en paramètre une liste et une fonction de comparaison
- et renvoie **True**si cette liste est triée par l'ordre défini par la fonction de comparaison, et renvoie **False** dans le cas contraire.

```
>>> est_trie([3,1,4,1,5,9,2], compare)
False
>>> est_trie([1,1,2,3,4,5,9], compare)
True
```

```
def est_triee(l:list[int], comp:Callable[[int,int],int])->bool:
    i=0
    while i<len(liste)-1 and comp([i],[i+1])<0:
        i+=1
    Return i==len((liste).
```

2 Ordre sur les dates

On suppose dans cet exercice que les dates sont représentées par des objets d'une classe **Date** possédant trois attributs nommées **'jour'** **'mois'** et **'annee'**. On suppose de plus que les valeurs associées à ces attributs sont des entiers.

On donne la variable **etudiants** contenant des objets de type **Etudiant** possédant quatre attributs :, **nip** (un entier) , **nom** (une chaine), **prenom** (une chaine), **date_de_naissance** (de classe **Date**) :

```
etudiants = [Etudiant(99998132, "Calbuth", "Raymond", Date(12, 12, 1987)),
   Etudiant(99994451, "Talon", "Achille", Date(7, 11, 1963)),
   Etudiant(99996348, "Calbuth", "Monique", Date(29, 7, 1987)),
   Etudiant(99995433, "Blanc-Sec", "Adèle", Date(17, 4, 1976)),
   Etudiant(99997674, "Brisefer", "Benoît", Date(15, 12, 1960)),
   Etudiant(99998324, "Lagaffe", "Gaston", Date(28, 2, 1957))]
```

2. Donnez une expression utilisant la fonction **sorted** égale à une nouvelle liste contenant les mêmes données que la liste **etudiants** mais rangées par ordre décroissant de nip.
3. Donnez une instruction utilisant la méthode **sort** permettant de modifier la variable **etudiants** afin que la liste des étudiants soit rangée par ordre croissant des nip.
4. Donnez une instruction permettant de modifier la variable **etudiants** afin que la liste des étudiants soit rangée par ordre décroissant des nip.
5. Donnez une instruction permettant de modifier la variable **etudiants** afin que la liste des étudiants soit rangée par ordre croissant de nom.
6. Proposez une fonction de comparaison d'étudiants avec laquelle vous pourrez modifier la variable **etudiants** afin que la liste des étudiants soit rangée par ordre croissant de date de naissance.

2. sorted(etudiants, True, key = lambda x : x.nip).
3. etudiants.sort(reverse = False, key = lambda x : x.nip).
4. etudiants.sort(reverse = True, key = lambda x : x.nip).
5. etudiants.sort(reverse = False, key = lambda x : x.nom)

3 Ordre sur les chaines

Dans cet exercice, on souhaite définir une variante de l'ordre des caractères puis étendre cette variante aux chaines de caractères. On dit que c est une lettre si c est

- soit entre 'A' et 'Z'
- soit entre 'a' et 'z'

1. Réalisez une fonction `compare_car` paramétrée par deux caractères telle que `compare_car(c1,c2)`
   - renvoie soit -1, 0 ou 1
   - avec pour les lettres l'ordre défini par `'aAbBcC....yYzZ'`
   - tout caractère non lettre étant plus grand que les lettres, et dans leur ordre habituel.

```
>>> compare_car('a', 'A')
-1
>>> compare_car('A','b')
-1
>>> compare_car('@', 'Z')
1
>>> compare_car('9', '@')
-1
```

65 -> A -> 65 x2 -> 65 - 92

97 -> a 97 -> 123

```
def get_ord_perso(c1):
    if 97 <= c1 < 123:
        Return (ord(c1) - 97) * 2
    elif 65 <= c1 < 91:
        Return (ord(c1) - 64) * 2
    else:
        Return ord(c1)
```

```
def compare_car(c1:str, c2:str) -> int:
    ord_c1 = get_ord_perso(c1)
    ord_c2 = get_ord_perso(c2)
    if ord_c1 == ord_c2:
        return 0
    elif ord_c1 < ord_c2:
        return -1
    else:
        return 1.
```

2. Réalisez une fonction `compare_chaine` permettant de comparer les chaines en utilisant la fonction `compare_car` précédente, de sorte que

- les chaînes soient rangées par longueur croissante ;
- les chaînes de même longueurs soient rangées par ordre lexicographique induit par `compare_car`.

```
>>> compare_chaine('','Tim')
-1
>>> compare_chaine('Timoleon', 'TiM')
1
>>> compare_chaine('Timoleon', 'TimoLeon')
-1
>>> compare_chaine('TimoLeon', 'TimoLeon')
0
```

```
def compare_chaine(s1:str, s2:str)->int:
    if len(s1) != len(s2):
        return 1 if len(s1) > len(s2) else -1.
    else:
        i = 0
        while i < len(s1) and compare_car(s1[i], s2[i]) == 0:
            i+=1
        return 1 if compare_car(s1[i], s2[i]) > 0 else -1.
```

4 Retour sur les doublons

1. Proposez un algorithme utilisant les tris pour construire la liste des doublons d'une liste donnée.
2. Comparez le nombre de comparaisons d'éléments de la liste effectuées par cet algorithme avec celui de l'algorithme que vous avez proposé dans l'exercice de la feuille précédente.

1.
```
def get_doublons(l:list)->list:
    Res = []
    l.sort()
    for i in range(len(l)-1):
        if comp(l[i], l[i+1]) == 0 and l[i] not in Res:
            Res.append(l[i])
    Return Res.
```

5 Variante du tri par sélection

Nous avons présenté le tri par sélection du plus petit élément de la tranche restant à trier. Il est possible aussi de faire un tri par sélection du plus grand élément.

1. Donnez l'algorithme de tri par sélection du plus grand élément.
2. Implantez cet algorithme pour réaliser une procédure `tri_sel_max` qui trie de cette manière la liste passée en paramètre.

1.
```
def sel_max(l, a, b):
    ind_max = b
    for i in range(a+1, b-1):
        if l[i] > l[ind_max]
            ind_max = i
    Return ind_max.
```

2.
```
def tri_sel_max(l, a, b)
    for i in range(n-2, 0, -1):
        max = sel_max(l, i, n)
        l[i], l[max] = l[max], l[i].
```