Python
Exercices supplémentaires

Exercice 1
Écrire une fonction `meme_parite` qui prend en argument une liste d'entiers et qui retourne une liste de booléens de même longueur. La valeur de chaque booléen à un indice donné est `True` si l'élément à cet indice dans la liste donnée est de même parité que l'indice lui-même, `False` sinon.

```
>>> meme_parite([1, 3, 2, 4, 6])
[False, True, True, False, True]
```

Exercice 2
On dit qu'une liste d'entiers est *spécifique* si elle contient au moins un entier de chacune de ces quatre catégories :

- un entier pair
- un entier multiple de 5
- un entier entre 0 et 9
- un entier strictement négatif

Un entier peut tout à fait à lui seul appartenir à deux ou plus de ces catégories.
Écrire une fonction `est_specifique` qui prend en argument une liste d'entiers et qui retourne un booléen : `True` si la liste est spécifique, `False` sinon.

```
>>> est_specifique([1, 3, 5, 4, 6])
False
>>> est_specifique([6, -1, 15, 13])
True
```

Exercice 3
Écrire une fonction `demi_moitie` qui prend en argument une liste d'entiers et qui retourne une liste d'entiers, qui contient un entier sur deux de la deuxième moitié de la liste.
Si la liste a un nombre impair d'éléments, la liste retournée commence par l'élément du milieu.
Si la liste a un nombre pair d'éléments, la liste retournée commence par le premier élément de la moitié droite.

```
>>> demi_moitie([1, 3, 2, 4, 6])
[2, 6]
>>> demi_moitie([1, 3, 2, 4, 6, 5])
[4, 5]
```

Exercice 4
Écrire une fonction `echange_paires` qui prend en argument une liste d'entiers et qui retourne une liste d'entiers dont les éléments sont ceux de la liste échangés paire après paire. Si la liste est impaire, le dernier élément reste tel quel.

```
>>> echange_paires([1, 3, 2, 4, 6, 5])
[3, 1, 4, 2, 5, 6]
>>> echange_paires([1, 3, 2, 4, 6])
[3, 1, 4, 2, 6]
>>> echange_paires([1])
[1]
```

Exercice 5
Écrire une fonction `signes_groupes` qui prend en argument une liste d'entiers et qui retourne le booléen `True` si tous les éléments positifs ou nuls sont côte à côte et tous les éléments négatifs ou nuls sont côte à côte.
Autrement dit, il ne faut au plus qu'un seul changement de signe dans les nombres (le 0 ne faisant pas changer de signe), éventuellement aucun.

```
>>> signes_groupes([-1, -3, -2, -4, 6, 5])
True
>>> signes_groupes([1, 3, 2, -4, -6])
True
>>> signes_groupes([1, 3, 2])
True
>>> signes_groupes([1, -3, 2, -4, -6])
False
```

Exercice 6
Écrire une fonction `compte_hausses_baisses` qui prend en argument une liste d'entiers et qui retourne un couple d'entiers indiquant le nombre de hausses puis le nombre de baisses dans la liste, en regardant l'évolution des valeurs successives de la liste. On considère à la fois comme hausse et comme baisse le cas où les valeurs successives sont égales.

```
>>> compte_hausses_baisses([1, 3, 2, -4, -6])
1,3
>>> compte_hausses_baisses([12, 15, 14, 14, 14, 10, 16, 15])
4,5
```

Exercice 7
Un *plateau* est une succession de valeurs égales dans une liste. La valeur du plateau est la valeur de ses éléments égaux.
Écrire une fonction `max_plateau` qui prend en argument une liste d'entiers et qui retourne un couple contenant la longueur du plateau le plus grand de la liste ainsi que la valeur de ce plateau (la première si d'autres plateau ont la même longueur).

```
>>> max_plateau([1, 2, 2, 2, 3, 3, 3, 3, 3, 8, 0, 0, 0, 0, 0, 4, 4])
5,3
```

On pourra également faire en sorte que la deuxième valeur du couple retourné soit la liste des valeurs des plateaux de même longueur maximale. Dans ce cas :

```
>>> max_plateau_bis([1, 2, 2, 2, 3, 3, 3, 3, 3, 8, 0, 0, 0, 0, 0, 4, 4])
5,[3,0]
```

Exercice 8
Écrire une fonction `somme_imbriquee` qui prend en argument une liste de listes d'entiers et qui retourne la somme des entiers contenus.

```
>>> somme_imbriquee([[1, 2], [3], [4, 5, 6]])
21
```

Exercice 9
Écrire une fonction `somme_cumulee` qui prend en argument une liste d'entiers et qui retourne une liste dont chaque élément d'indice `i` est la somme des éléments de la liste donnée en argument jusqu'à l'indice `i`.

```
>>> somme_cumulee([1, 2, 3])
[1, 3, 6]
```

Exercice 10

Écrire une fonction `possede_doublon` qui prend en argument une liste et qui retourne `True` si cette liste contient au moins un élément qui apparaît plus qu'une fois dans la liste, `False` sinon. La fonction ne doit pas modifier la liste donnée en argument.

```
>>> possede_doublon([1, 2, 3, 4])
False
>>> possede_doublon([1, 'a', 3, 'a'])
True
```

Exercice 11

Le fichier `mots.txt` fourni contient de très nombreux mots français, triés par ordre alphabétique. Le but de cet exercice est de récupérer ces mots et de les placer dans une liste dans laquelle on pourra effectuer des recherches.

1. Écrire une fonction `fichier_vers_liste` qui prend en argument l'adresse (sous forme de chaîne de caractère) d'un fichier contenant des mots (un mot par ligne) et qui retourne la liste de ces mots.

    chaque ligne lue dans le fichier contient le caractère de retour à la ligne, qu'il faut donc supprimer.

   On doit alors avoir :

   ```
   >>> liste_mots = fichier_vers_liste("mots.txt")
   >>> len(liste_mots)
   323578
   ```

   Aide : pour ouvrir et lire un fichier (texte), on peut utiliser la syntaxe suivante :

   ```
   with open("adresse_du_fichier") as fichier: # on ouvre le fichier
       for ligne in fichier: # on parcourt les lignes du fichier
           # action à effectuer sur chaque ligne
           # ligne est une chaîne de caractère
   ```

2. Écrire une fonction `est_triee` qui prend en argument une liste de chaînes de caractères et qui retourne `True` si la liste est bien triée par ordre alphabétique, `False` sinon.

    les tirets (`'-'`) ne doivent pas être considérés lors de la comparaison, il s'agit donc de les supprimer des mots à comparer.

   On vérifiera alors que :

   ```
   >>> est_triee(liste_mots)
   True
   ```

   Aide : pour remplacer toutes les occurrences d'une chaîne de caractère par une autre dans un texte, on peut utiliser la syntaxe suivante :

   ```
   texte.replace(chaine_a_trouver, chaine_de_remplacement)
   ```

   Exemple :

   ```
   >>> "ceci est un exemple".replace("e","y")
   'cyci yst un yxymply'
   ```

3. Écrire une fonction `est_present(mot,liste)` qui effectue la **recherche dichotomique** d'un `mot` dans une `liste` de mots triée par ordre alphabétique, et retourne `True` si le mot est présent, `False` sinon.

```
>>> est_present("coucou",liste_mots)
True
>>> est_present("pipe-line",liste_mots)
True
>>> est_present("noone",liste_mots)
False
```

Rappel : La fonction de recherche par dichotomie donnée en cours est la suivante :

```
def recherche_par_dichotomie(x,liste):
    gauche = 0
    droite = len(liste)
    while droite - gauche > 1:
        centre = (gauche + droite) // 2
        if x < liste[centre]:
            droite = centre
        else:
            gauche = centre
    if x == liste[gauche]:
        return gauche
```

 Cette fonction retourne un indice (celui de la valeur `x` cherchée dans `liste`).

Il est donc nécessaire de modifier le code pour qu'elle retourne un booléen, et tenir compte du problème des tirets déjà évoqué plus haut.