Exercices parcours de liste

1. Écrire une fonction `recherche_occurrence` prenant deux paramètres, une liste d'entiers `l` et un entier `n`, et qui parcourt tous les éléments de `l` à la recherche de la valeur `n`.
   Si `n` est présent dans `l`, alors la fonction renvoie `True`, sinon elle renvoie `False`.

   Pseudo-code :

   ```
   recherche_occurrence(liste L, entier n) -> booléen:
       On parcours tous les éléments de L :
           Si l'élément courant est égal à n :
               On retourne Vrai, sinon on continue.
       Le parcours n'a rien trouvé, on retourne Faux.
   ```

   Tests :

   ```
   assert recherche_occurrence([0,1,2,3,4], 5) == False
   assert recherche_occurrence([0,1,2,3,4], 2) == True
   ```

2. On souhaite écrire une fonction `somme` qui prenant en paramètre une liste d'entiers `l` et qui parcourt tous les éléments de `l` pour renvoyer la somme de ses éléments.
   1. Écrire deux cas de tests avec des assertions pour la fonction `somme`.
   2. Coder la fonction `somme`. (Aidez vous si besoin de l'exemple du cours.)

3. Écrire une fonction `produit` prenant en paramètre une liste d'entiers `l`, qui parcourt tous les éléments de `l` pour renvoyer le produit de ses éléments.
   1. Écrire deux cas de tests avec des assertions pour la fonction `produit`.
   2. Coder la fonction `produit`.
   3. Écrire une fonction `factoriel` prenant en paramètre un entier positif `n` et qui utilise la fonction `produit` pour calculer `n!`.
      Rappel : `n!` Est égal au produit des nombres de 1 à n. $n! = 1 \times 2 \times 3 \times ... \times (n-1) \times n$

4. Écrire une fonction `moyenne` prenant en paramètre une liste d'entiers `l` et qui parcourt tous les éléments de `l` pour renvoyer la moyenne de ses éléments. Tester la.

5. Écrire une fonction `minimum` prenant en paramètre une liste d'entiers `l` et qui parcourt tous les éléments de `l` pour renvoyer la valeur du plus petit de ses éléments. Tester la.

   Pseudo-code :

   ```
   minimum(liste L) -> entier:
       valmin := premier élément de L
       On parcours tous les éléments de L :
           Si l'élément courant est inférieur à valmin :
               valmin := l'élément courant
       On retourne valmin.
   ```

   Tests :

   ```
   assert minimum([2, 5, 3, 1, 4]) == 1
   assert minimum([-2, -5, -3, -1, -4]) == -5
   ```

6. Écrire une fonction `maximum` prenant en paramètre une liste d'entiers `l` et qui parcourt tous les éléments de `l` pour renvoyer la valeur du plus grand de ses éléments. Tester la.

7. Écrire une fonction `indice_min` qui prend en paramètre une liste d'entiers `l` et qui parcourt tous les éléments de `l` pour renvoyer l'indice du plus petit de ses éléments.

Pseudo-code :

```
indice_min(liste L) -> entier:
    indice := 0
    Pour i de 0 à taille(L) - 1 :
        Si L[i] est plus petit que L[indice] :
            indice := i
    On retourne indice.
```

Tests :

```
assert indice_min([2, 5, 3, 1, 4]) == 3
assert indice_min([-2, -5, -3, -1, -4]) == 1
```

8. Écrire une fonction `indice_max` qui prend en paramètre une liste d'entiers `l` et qui parcourt tous les éléments de `l` pour renvoyer l'indice du plus grand de ses éléments. Tester la.

9. Écrire une fonction `nombre_occurrence` prenant deux paramètres, une liste d'entiers `l` et un entier `n`, qui parcourt tous les éléments de `l` et renvoie le nombre d'occurrence de `n`.

10. Écrire une fonction `indice_premiere_occurrence` prenant deux paramètres, une liste d'entiers `l` et un entier `n`, qui parcourt tous les éléments de `l` et renvoie l'indice de la première occurrence de `n`. Tester la.
    1. De la même façon, écrire `indice_derniere_occurence` qui renvoie l'indice de la dernière occurrence.
    2. De la même façon, écrire `indice_occurence` qui renvoie une liste contenant tous les indices des occurrences de `n`.

Bonus

1. Écrivez une fonction `doublon(l)` qui renvoie `True` s'il existe (au moins) un doublon dans `l` et `False` dans le cas contraire.

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

2. Écrivez une fonction `miroir(l)` qui reçoit une liste `l` en argument et la modifie pour échanger le premier élément avec le dernier, le deuxième avec l'avant-dernier, etc. Dit autrement, on remplace le tableau par son image miroir.

```
>>> tab = [1, 2, 3, 4]
>>> miroir(tab)
>>> tab
[4, 3, 2, 1]
```