EXERCICE 1 (10 points)

Écrire une fonction `compte_occurrences` prenant en paramètres une valeur `x` et un tableau `tab` (de type `list`) et renvoyant le nombre d'occurrences de `x` dans `tab`.

L'objectif de cet exercice étant de parcourir un tableau, il est interdit d'utiliser la méthode `count` des listes Python.

Exemples :

```
>>> compte_occurrences(5, [])
0
>>> compte_occurrences(5, [-2, 3, 1, 5, 3, 7, 4])
1
>>> compte_occurrences('a', ['a','b','c','a','d','e','a'])
3
```

EXERCICE 1 (10 points)

Dans cet exercice on cherche à calculer la moyenne pondérée d'un élève dans une matière donnée. Chaque note est associée à un coefficient qui la pondère.

Par exemple, si ses notes sont : 14 avec coefficient 3, 12 avec coefficient 1 et 16 avec coefficient 2, sa moyenne pondérée sera donnée par

$$\frac{14 \times 3 + 12 \times 1 + 16 \times 2}{3 + 1 + 2} = 14,333...$$

Écrire une fonction `moyenne` :

- qui prend en paramètre une liste `notes` non vide de tuples à deux éléments entiers de la forme `(note, coefficient)` (`int` ou `float`) positifs ou nuls ;
- et qui renvoie la moyenne pondérée des notes de la liste sous forme de flottant si la somme des coefficients est non nulle, `None` sinon.

Exemple :

```
>>> moyenne([(8, 2), (12, 0), (13.5, 1), (5, 0.5)])
9.142857142857142
>>> moyenne([(3, 0), (5, 0)])
None
```

EXERCICE 2 (10 points)

On souhaite programmer une fonction indiquant le point le plus proche d'un point de départ dans un tableau de points non vide. Les points sont tous à coordonnées entières et sont donnés sous la forme d'un tuple de deux entiers. Le tableau des points à traiter est donc un tableau de tuples.

On rappelle que la distance d entre deux points du plan de coordonnées $(x; y)$ et $(x'; y')$ vérifie la formule :

$$d^2 = (x - x')^2 + (y - y')^2$$

Compléter le code des fonctions `distance_carre` et `point_le_plus_proche` fournies ci-dessous pour qu'elles répondent à leurs spécifications.

```
def distance_carre(point1, point2):
    """ Calcule et renvoie la distance au carre entre
    deux points."""
    return (...)**2 + (...)**2

def point_le_plus_proche(depart, tab):
    """ Renvoie les coordonnées du premier point du tableau tab se
    trouvant à la plus courte distance du point depart."""
    min_point = tab[0]
    min_dist = ...
    for i in range(1, len(tab)):
        if distance_carre(tab[i], depart) < ...:
            min_point = ...
            min_dist = ...
    return min_point
```

Exemples :

```
>>> distance_carre((1, 0), (5, 3))
25
>>> distance_carre((1, 0), (0, 1))
2
>>> point_le_plus_proche((0, 0), [(7, 9), (2, 5), (5, 2)])
(2, 5)
>>> point_le_plus_proche((5, 2), [(7, 9), (2, 5), (5, 2)])
(5, 2)
```

EXERCICE 1 (10 points)

Écrire une fonction `couples_consecutifs` qui prend en paramètre un tableau de nombres entiers `tab` non vide (type `list`), et qui renvoie la liste Python (éventuellement vide) des couples d'entiers consécutifs successifs qu'il peut y avoir dans `tab`.

Exemples :

```
>>> couples_consecutifs([1, 4, 3, 5])
[]
>>> couples_consecutifs([1, 4, 5, 3])
[(4, 5)]
>>> couples_consecutifs([1, 1, 2, 4])
[(1, 2)]
>>> couples_consecutifs([7, 1, 2, 5, 3, 4])
[(1, 2), (3, 4)]
>>> couples_consecutifs([5, 1, 2, 3, 8, -5, -4, 7])
[(1, 2), (2, 3), (-5, -4)]
```

EXERCICE 1 (10 points)

On considère des chaînes de caractères contenant uniquement des majuscules et des caractères `*` appelées *mots à trous*.

Par exemple `INFO*MA*IQUE`, `***I***E**` et `*S*` sont des mots à trous.

Programmer une fonction `correspond` :

- qui prend en paramètres deux chaînes de caractères `mot` et `mot_a_trous` où `mot_a_trous` est un mot à trous comme indiqué ci-dessus ;
- et qui renvoie :
  - `True` si on peut obtenir `mot` en remplaçant convenablement les caractères `'*'` de `mot_a_trous` ;
  - `False` sinon.

Exemple :

```
>>> correspond('INFORMATIQUE', 'INFO*MA*IQUE')
True
>>> correspond('AUTOMATIQUE', 'INFO*MA*IQUE')
False
>>> correspond('STOP', 'S*')
False
>>> correspond('AUTO', '*UT*')
True
```

EXERCICE 2 (10 points)

On considère au plus 26 personnes A, B, C, D, E, F … qui peuvent s'envoyer des messages avec deux règles à respecter :

- chaque personne ne peut envoyer des messages qu'à une seule personne (éventuellement elle-même),
- chaque personne ne peut recevoir des messages qu'en provenance d'une seule personne (éventuellement elle-même).

Voici un exemple - avec 6 personnes - de « plan d'envoi des messages » qui respecte les règles ci-dessus, puisque chaque personne est présente une seule fois dans chaque colonne :

- A envoie ses messages à E
- E envoie ses messages à B
- B envoie ses messages à F
- F envoie ses messages à A
- C envoie ses messages à D
- D envoie ses messages à C

Le dictionnaire correspondant à ce plan d'envoi est alors le suivant :

```
plan_a = {'A':'E', 'B':'F', 'C':'D', 'D':'C', 'E':'B', 'F':'A'}
```

Un cycle est une suite de personnes dans laquelle la dernière est la même que la première.

Sur le plan d'envoi `plan_a` des messages ci-dessus, il y a deux cycles distincts : un premier cycle avec A, E, B, F et un second cycle avec C et D.

En revanche, le plan d'envoi `plan_b` ci-dessous :

```
plan_b = {'A':'C', 'B':'F', 'C':'E', 'D':'A', 'E':'B', 'F':'D'}
```

comporte un unique cycle : A, C, E, B, F, D. Dans ce cas, lorsqu'un plan d'envoi comporte un *unique cycle*, on dit que le plan d'envoi est *cyclique*.

Pour savoir si un plan d'envoi de messages comportant N personnes est cyclique, on peut utiliser l'algorithme ci-dessous :

- on part d'un expéditeur (ici A) et on inspecte son destinataire dans le plan d'envoi,
- chaque destinataire devient à son tour expéditeur, selon le plan d'envoi, tant qu'on ne « retombe » pas sur l'expéditeur initial,
- le plan d'envoi est cyclique si on l'a parcouru en entier.

Compléter la fonction `est_cyclique` située à la page suivante en respectant la spécification. On rappelle que la fonction Python `len` permet d'obtenir la longueur d'un dictionnaire.

```
def est_cyclique(plan):
    '''Prend en paramètre un dictionnaire `plan` correspondant à
    un plan d'envoi de messages (ici entre les personnes A, B, C,
    D, E, F).
    Renvoie True si le plan d'envoi de messages est cyclique et
    False sinon.'''
    expediteur = 'A'
    destinataire = plan[...]
    nb_destinataires = 1

    while destinataire != expediteur:
        destinataire = ...
        nb_destinataires = ...

    return nb_destinataires == ...
```

Exemples :

```
>>> est_cyclique({'A':'E','F':'A','C':'D','E':'B','B':'F','D':'C'})
False
>>> est_cyclique({'A':'E','F':'C','C':'D','E':'B','B':'F','D':'A'})
True
>>> est_cyclique({'A':'B','F':'C','C':'D','E':'A','B':'F','D':'E'})
True
>>> est_cyclique({'A':'B','F':'A','C':'D','E':'C','B':'F','D':'E'})
False
```

EXERCICE 2 (10 points)

On considère dans cet exercice un algorithme glouton pour le rendu de monnaie. Pour rendre une somme en monnaie, on utilise à chaque fois la plus grosse pièce possible et ainsi de suite jusqu'à ce que la somme restante à rendre soit nulle.

Les pièces de monnaie utilisées sont :

```
pieces = [1, 2, 5, 10, 20, 50, 100, 200]
```

On souhaite écrire une fonction `rendu_monnaie` qui prend en paramètres

- un entier `somme_due` représentant la somme à payer ;
- un entier `somme_versee` représentant la somme versée qui est supérieure ou égale à `somme_due` ;
- et qui renvoie un tableau de type `list` contenant les pièces qui composent le rendu de la monnaie restante, c'est-à-dire de `somme_versee - somme_due`.

Ainsi, l'instruction `rendu_monnaie(452, 500)` renvoie le tableau `[20, 20, 5, 2, 1]`.

En effet, la somme à rendre est de 48 euros soit `20 + 20 + 5 + 2 + 1`.

Le code de la fonction `rendu_monnaie` est donné ci-dessous :

```
def rendu_monnaie(somme_due, somme_versee):
    '''Renvoie la liste des pièces à rendre pour rendre la monnaie
    lorsqu'on doit rendre somme_versee - somme_due'''
    rendu = ...
    a_rendre = ...
    i = len(pieces) - 1
    while a_rendre > ...:
        while pieces[i] > a_rendre:
            i = i - 1
        rendu.append(...)
        a_rendre = ...
    return rendu
```

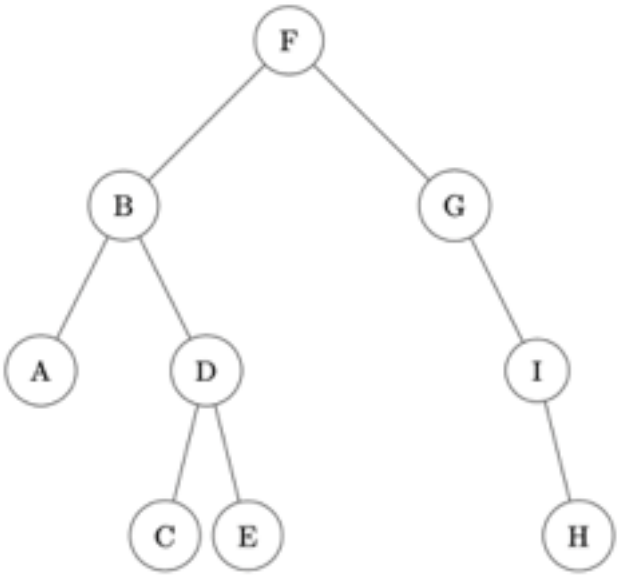
Compléter ce code et le tester :

```
>>> rendu_monnaie(700, 700)
[]
>>> rendu_monnaie(102, 500)
[200, 100, 50, 20, 20, 5, 2, 1]
```

EXERCICE 1 (10 points)

Dans cet exercice, un arbre binaire de caractères non vide est stocké sous la forme d'un dictionnaire où les clefs sont les caractères des nœuds de l'arbre et les valeurs, pour chaque clef, la liste des caractères des fils gauche et droit du nœud. On utilise la valeur `''` pour représenter un fils vide.

Par exemple, l'arbre



est stocké dans

```
a = {'F':['B','G'], 'B':['A','D'], 'A':['',''], 'D':['C','E'], \
     'C':['',''], 'E':['',''], 'G':['','I'], 'I':['','H'], \
     'H':['','']}
```

Écrire une fonction récursive `taille` prenant en paramètres un arbre binaire `arbre` non vide sous la forme d'un dictionnaire et un caractère `lettre` qui est la valeur du sommet de l'arbre, et qui renvoie la taille de l'arbre à savoir le nombre total de nœuds.

On observe que, par exemple, `arbre[lettre][0]`, respectivement `arbre[lettre][1]`, permet d'atteindre la clé du sous-arbre gauche, respectivement droit, de l'arbre `arbre` de sommet `lettre`.

Exemples :

```
>>> taille(a, 'F')
9
>>> taille(a, 'B')
5
>>> taille(a, 'I')
2
```

EXERCICE 2 (10 points)

On considère l'algorithme de tri de tableau suivant : à chaque étape, on parcourt le sous-tableau des éléments non rangés et on place le plus petit élément en première position de ce sous-tableau.

Exemple avec le tableau : `t = [41, 55, 21, 18, 12, 6, 25]`

- Étape 1 : on parcourt tous les éléments du tableau, on permute le plus petit élément avec le premier.

  Le tableau devient `t = [`**`6`**`, 55, 21, 18, 12, 41, 25]`

- Étape 2 : on parcourt tous les éléments **sauf le premier**, on permute le plus petit élément trouvé avec le second.

  Le tableau devient : `t = [`**`6, 12`**`, 21, 18, 55, 41, 25]`

Et ainsi de suite.

Le programme ci-dessous implémente cet algorithme.

```
def echange(tab, i, j):
    '''Echange les éléments d'indice i et j dans le tableau tab.'''
    temp = ...
    tab[i] = ...
    tab[j] = ...

def tri_selection(tab):
    '''Trie le tableau tab dans l'ordre croissant
    par la méthode du tri par sélection.'''
    N = len(tab)
    for k in range(...):
        imin = ...
        for i in range(..., N):
            if tab[i] < ...:
                imin = i
        echange(tab, ..., ...)
```

Compléter ce code de façon à obtenir :

```
>>> tab = [41, 55, 21, 18, 12, 6, 25]
>>> tri_selection(tab)
>>> tab
[6, 12, 18, 21, 25, 41, 55]
```

EXERCICE 1 (10 points)

Programmer la fonction `recherche`, prenant en paramètre un tableau non vide `tab` (type `list`) d'entiers et un entier `n`, et qui renvoie l'indice de la **dernière** occurrence de l'élément cherché. Si l'élément n'est pas présent, la fonction renvoie `None`.

Exemples

```
>>> recherche([5, 3],1) # renvoie None
>>> recherche([2,4],2)
0
>>> recherche([2,3,5,2,4],2)
3
```