EXERCICE 1 (4 points)

Programmer la fonction `verifie` qui prend en paramètre un tableau de valeurs numériques non vide et qui renvoie `True` si ce tableau est trié dans l'ordre croissant, `False` sinon.

Exemples :

```
>>> verifie([0, 5, 8, 8, 9])
True
>>> verifie([8, 12, 4])
False
>>> verifie([-1, 4])
True
>>> verifie([5])
True
```

EXERCICE 2 (4 points)

Les résultats d'un vote ayant trois issues possibles 'A', 'B' et 'C' sont stockés dans un tableau.

Exemple :

```
Urne = ['A', 'A', 'A', 'B', 'C', 'B', 'C', 'B', 'C', 'B']
```

La fonction `depouille` doit permettre de compter le nombre de votes exprimés pour chacune des issues. Elle prend en paramètre un tableau et renvoie le résultat dans un dictionnaire dont les clés sont les noms des issues et les valeurs le nombre de votes en leur faveur.

La fonction `vainqueur` doit désigner le nom du ou des gagnants. Elle prend en paramètre un dictionnaire dont la structure est celle du dictionnaire renvoyé par la fonction `depouille` et renvoie un tableau. Ce tableau peut donc contenir plusieurs éléments s'il y a des ex-aequo.

Compléter les fonctions `depouille` et `vainqueur` fournies à la page suivante pour qu'elles renvoient les résultats attendus.

```
def depouille(urne):
    resultat = ...
    for bulletin in urne:
        if ...:
            resultat[bulletin] = resultat[bulletin] + 1
        else:
            ...
    return resultat


def vainqueur(election):
    vainqueur = ''
    nmax = 0
    for candidat in election:
        if ... > ... :
            nmax = ...
            vainqueur = candidat
    liste_finale = [nom for nom in election if election[nom] == ...]
    return ...
```

Exemples d'utilisation :

```
>>> election = depouille(urne)
>>> election
{'B': 4, 'A': 3, 'C': 3}

>>> vainqueur(election)
['B']
```

EXERCICE 1 (4 points)

Écrire une fonction `indices_maxi` qui prend en paramètre une liste `tab`, non vide, de nombres entiers et renvoie un couple donnant d'une part le plus grand élément de cette liste et d'autre part la liste des indices de la liste `tab` où apparaît ce plus grand élément.

Exemples :

```
>>> indices_maxi([1, 5, 6, 9, 1, 2, 3, 7, 9, 8])
(9, [3, 8])

>>> indices_maxi([7])
(7, [0])
```

EXERCICE 2 (4 points)

Cet exercice utilise des piles qui seront représentées en Python par des listes (de type `list`).

On rappelle que l'expression `liste_1 = list(liste)` fait une copie de `liste` indépendante de `liste`, que l'expression `x = liste.pop()` enlève le sommet de la pile `liste` et le place dans la variable `x` et, enfin, que l'expression `liste.append(v)` place la valeur `v` au sommet de la pile `liste`.

Compléter le code Python de la fonction `positif` ci-dessous qui prend une pile `liste` de nombres entiers en paramètre et qui renvoie la pile des entiers positifs dans le même ordre, sans modifier la variable `liste`.

```
def positif(pile):
    pile_1 = ...(pile)
    pile_2 = ...
    while pile_1 != []:
        x = ...
        if ... >= 0:
            pile_2.append(...)
    while pile_2 != ...:
        x = pile_2.pop()
        ...
    return pile_1
```

Exemples :

```
>>> positif([-1, 0, 5, -3, 4, -6, 10, 9, -8])
[0, 5, 4, 10, 9]

>>> positif([-2])
[]
```

EXERCICE 1 (4 points)

Dans cet exercice, les nombres sont des entiers ou des flottants.

Écrire une fonction `moyenne` renvoyant la moyenne pondérée d'une liste non vide, passée en paramètre, de tuples à deux éléments de la forme `(valeur, coefficient)` où `valeur` et `coefficient` sont des nombres positifs ou nuls.
Si la somme des coefficients est nulle, la fonction renvoie `None`, si la somme des coefficients est non nulle, la fonction renvoie, sous forme de flottant, la moyenne des valeurs affectées de leur coefficient.

Exemples :

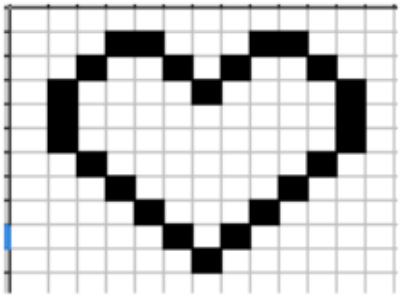
```
>>> moyenne([(8, 2), (12, 0), (13.5, 1), (5, 0.5)])
9.142857142857142
>>> moyenne([(3, 0), (5, 0)])
None
```

Dans le premier exemple la moyenne est calculée par la formule :

$$\frac{8 \times 2 + 12 \times 0 + 13{,}5 \times 1 + 5 \times 0{,}5}{2 + 0 + 1 + 0{,}5}$$

EXERCICE 2 (4 points)

On travaille sur des dessins en noir et blanc obtenus à partir de pixels noirs et blancs :



La figure « cœur » ci-contre va servir d'exemple.

On la représente par une grille de nombres, c'est-à-dire par une liste composée de sous-listes de mêmes longueurs.

Chaque sous-liste représentera donc une ligne du dessin.

Dans le code ci-dessous, la fonction `affiche` permet d'afficher le dessin. Les pixels noirs (1 dans la grille) seront représentés par le caractère `" *"` et les blancs (0 dans la grille) par deux espaces.

La fonction `zoomListe` prend en argument une liste `liste_depart` et un entier `k`. Elle renvoie une liste où chaque élément de `liste_depart` est dupliqué `k` fois.

La fonction `zoomDessin` prend en argument la grille `dessin` et renvoie une grille où toutes les lignes de `dessin` sont zoomées `k` fois (c'est-à-dire, on applique à chaque ligne la fonction `zoomListe` avec comme second paramètre `k`) et répétées `k` fois.

Compléter le code ci-dessous :

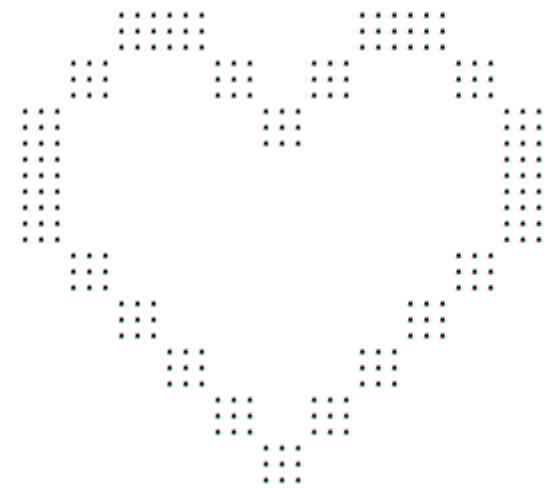
```
coeur = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0],
         [0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0],
         [0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0],
         [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
         [0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0],
         [0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0],
         [0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0],
         [0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

def affiche(dessin):
    ''' affichage d'une grille : les 1 sont représentés par
        des " *" , les 0 par deux espaces "  ".
        La valeur "" donnée au paramètre end permet de ne pas avoir
        de saut de ligne.'''
    for ligne in dessin:
        for col in ligne:
            if col == 1:
                print(" *", end= "")
            else:
                print("  ", end= "")
        print()
```

```
def zoomListe(liste_depart, k):
    '''renvoie une liste contenant k fois chaque
       élément de liste_depart'''
    liste_zoom = ...
    for elt in ... :
        for i in range(k):
            ...
    return liste_zoom

def zoomDessin(grille, k):
    '''renvoie une grille où les lignes sont zoomées k fois
       ET répétées k fois'''
    grille_zoom = []
    for elt in grille:
        liste_zoom = ...
        for i in range(k):
            ... .append(...)
    return grille_zoom
```

Résultats à obtenir :

```
>>> affiche(coeur)
```

```
>>> affiche(zoomDessin(coeur, 3))
```



EXERCICE 1 (4 points)

Écrire une fonction `a_doublon` qui prend en paramètre une liste **triée** de nombres et renvoie `True` si la liste contient au moins deux nombres identiques, `False` sinon.

Par exemple :

```
>>> a_doublon([])
False

>>> a_doublon([1])
False

>>> a_doublon([1, 2, 4, 6, 6])
True

>>> a_doublon([2, 5, 7, 7, 7, 9])
True

>>> a_doublon([0, 2, 3])
False
```

EXERCICE 2 (4 points)

On souhaite générer des grilles du jeu de démineur à partir de la position des bombes à placer.

On se limite à la génération de grilles carrées de taille $n \times n$ où n est le nombre de bombes du jeu.

Dans le jeu du démineur, chaque case de la grille contient soit une bombe, soit une valeur qui correspond aux nombres de bombes situées dans le voisinage direct de la case (au-dessus, en dessous, à droite, à gauche ou en diagonal : chaque case a donc 8 voisins si elle n'est pas située au bord de la grille).

Voici un exemple de grille 5x5 de démineur dans laquelle la bombe est représentée par une étoile :

1	1	1	0	0
1	*	1	1	1
2	2	3	2	*
1	*	2	*	3
1	1	2	2	*

On utilise une liste de listes pour représenter la grille et on choisit de coder une bombe par la valeur -1.

L'exemple ci-contre sera donc codé par la liste :

```
[[1,  1,  1,  0,  0],
 [1, -1,  1,  1,  1],
 [2,  2,  3,  2, -1],
 [1, -1,  2, -1,  3],
 [1,  1,  2,  2, -1]]
```

Compléter le code suivant afin de générer des grilles de démineur, on pourra vérifier que l'instruction `génère_grille([(1, 1), (2, 4), (3, 1), (3, 3), (4, 4)])` produit bien la liste donnée en exemple.

```
def voisinage(n, ligne, colonne):
    """ Renvoie la liste des coordonnées des voisins de la case
       (ligne, colonne) en gérant les cases sur les bords. """
    voisins = []
    for l in range(max(0,ligne-1), min(n, ligne+2)):
        for c in range(max(0, colonne-1), min(n, colonne+2)):
            if (l, c) != (ligne, colonne):
                voisins.append((l,c))
    return voisins

def incremente_voisins(grille, ligne, colonne):
    """ Incrémente de 1 toutes les cases voisines d'une bombe."""
    voisins = ...
    for l, c in voisins:
        if grille[l][c] != ...: # si ce n'est pas une bombe
            ...                 # on ajoute 1 à sa valeur

def genere_grille(bombes):
    """ Renvoie une grille de démineur de taille nxn où n est
        le nombre de bombes, en plaçant les bombes à l'aide de
        la liste bombes de coordonnées (tuples) passée en
        paramètre. """
    n = len(bombes)
    # Initialisation d'une grille nxn remplie de 0
    grille = [[0 for colonne in range(n)] for ligne in range(n)]
    # Place les bombes et calcule les valeurs des autres cases
    for ligne, colonne in bombes:
        grille[ligne][colonne] = ...  # place la bombe
        ...                           # incrémente ses voisins
    return grille
```

EXERCICE 1 (4 points)

Écrire en python deux fonctions :

- `lancer` de paramètre `n`, un entier positif, qui renvoie un tableau de type `list` de `n` entiers obtenus aléatoirement entre 1 et 6 (1 et 6 inclus) ;
- `paire_6` de paramètre `tab`, un tableau de type `list` de `n` entiers entre 1 et 6 obtenus aléatoirement, qui renvoie un booléen égal à `True` si le nombre de 6 est supérieur ou égal à 2, `False` sinon.

On pourra utiliser la fonction `randint(a,b)` du module `random` pour laquelle la documentation officielle est la suivante :

```
Renvoie un entier aléatoire N tel que a <=N <= b.
```

Exemples :

```
>>> lancer1 = lancer(5)
[5, 6, 6, 2, 2]
>>> paire_6(lancer1)
True
>>> lancer2 = lancer(5)
[6, 5, 1, 6, 6]
>>> paire_6(lancer2)
True
>>> lancer3 = lancer(3)
[2, 2, 6]
>>> paire_6(lancer3)
False
>>> lancer4 = lancer(0)
[]
>>> paire_6(lancer4)
False
```

EXERCICE 2 (4 points)

On considère une image en 256 niveaux de gris que l'on représente par une grille de nombres, c'est-à-dire une liste composée de sous-listes toutes de longueurs identiques.

La largeur de l'image est donc la longueur d'une sous-liste et la hauteur de l'image est le nombre de sous-listes.

Chaque sous-liste représente une ligne de l'image et chaque élément des sous-listes est un entier compris entre 0 et 255, représentant l'intensité lumineuse du pixel.

Le négatif d'une image est l'image constituée des pixels `x_n` tels que

`x_n + x_i = 255` où `x_i` est le pixel correspondant de l'image initiale.

Compléter le programme proposé page suivante :

```
def nbLig(image):
    '''renvoie le nombre de lignes de l'image'''
    return ...

def nbCol(image):
    '''renvoie la largeur de l'image'''
    return ...

def negatif(image):
    '''renvoie le négatif de l'image sous la forme
       d'une liste de listes'''

    # on créé une image de 0 aux mêmes dimensions que le paramètre
image
    L = [[0 for k in range(nbCol(image))] for i in range(nbLig(image))]

    for i in range(nbLig(image)):
        for j in range(...):
            L[i][j] = ...
    return L

def binaire(image, seuil):
    '''renvoie une image binarisée de l'image sous la forme
       d'une liste de listes contenant des 0 si la valeur
       du pixel est strictement inférieure au seuil
       et 1 sinon'''

    # on crée une image de 0 aux mêmes dimensions que le paramètre
image
    L = [[0 for k in range(nbCol(image))] for i in range(nbLig(image))]

    for i in range(nbLig(image)):
        for j in range(...):
            if image[i][j] < ... :
                L[i][j] = ...
            else:
                L[i][j] = ...
    return L
```

Exemples :

```
>>> img=[[20, 34, 254, 145, 6], [23, 124, 237, 225, 69], [197, 174,
207, 25, 87], [255, 0, 24, 197, 189]]
>>> nbLig(img)
4
>>> nbCol(img)
5
>>> negatif(img)
[[235, 221, 1, 110, 249], [232, 131, 18, 30, 186], [58, 81, 48, 230,
168], [0, 255, 231, 58, 66]]
>>> binaire(img,120)
[[0, 0, 1, 1, 0], [0, 1, 1, 1, 0], [1, 1, 1, 0, 0], [1, 0, 0, 1, 1]]
```

EXERCICE 1 (4 points)

Programmer la fonction `recherche`, prenant en paramètre un tableau non vide `tab` (de 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 la longueur du tableau.

Exemples :

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

EXERCICE 2 (4 points)

On souhaite programmer une fonction donnant la distance la plus courte entre un point de départ et une liste de points. Les points sont tous à coordonnées entières.

Les points sont donnés sous la forme d'un tuple de deux entiers.

La liste des points à traiter est donc un tableau, non vide, de tuples.

On rappelle que la distance entre deux points du plan de coordonnées (x ; y) et (x' ; y') est donnée par la formule :

$$d = \sqrt{(x - x')^2 + (y - y')^2}.$$

On importe pour cela la fonction racine carrée (`sqrt`) du module `math` de Python.

Compléter le code des fonctions `distance` et `plus_courte_distance` fournies à la page suivante pour qu'elles répondent à leurs spécifications.

```
from math import sqrt   # import de la fonction racine carrée

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

def plus_courte_distance(tab, depart):
    """ Renvoie le point du tableau tab se trouvant à la plus
    courte distance du point depart."""
    point = tab[0]
    min_dist = ...
    for i in range (1, ...):
        if distance(tab[i], depart)...:
            point = ...
            min_dist = ...
    return point
```

Exemples :

```
>>> distance((1, 0), (5, 3))
5.0
>>> distance((1, 0), (0, 1))
1.4142135623730951

>>> plus_courte_distance([(7, 9), (2, 5), (5, 2)], (0, 0))
(2, 5)
>>> plus_courte_distance([(7, 9), (2, 5), (5, 2)], (5, 2))
(5, 2)
```

EXERCICE 1 (4 points)

Programmer la fonction `fusion` prenant en paramètres deux tableaux non vides `tab1` et `tab2` (de type `list`) d'entiers, chacun dans l'ordre croissant, et renvoyant un tableau trié dans l'ordre croissant et contenant l'ensemble des valeurs de `tab1` et `tab2`.

Exemples :

```
>>> fusion([3, 5], [2, 5])
[2, 3, 5, 5]
>>> fusion([-2, 4], [-3, 5, 10])
[-3, -2, 4, 5, 10]
>>> fusion([4], [2, 6])
[2, 4, 6]
```

EXERCICE 2 (4 points)

Le but de cet exercice est d'écrire une fonction récursive `traduire_romain` qui prend en paramètre une chaîne de caractères, non vide, représentant un nombre écrit en chiffres romains et qui renvoie son écriture décimale.

Les chiffres romains considérés sont : I, V, X, L, C, D et M. Ils représentent respectivement les nombres 1, 5, 10, 50, 100, 500, et 1000 en base dix.

On dispose d'un dictionnaire `romains` dont les clés sont les caractères apparaissant dans l'écriture en chiffres romains et les valeurs sont les nombres entiers associés en écriture décimale :

```
romains = {"I":1, "V":5, "X":10, "L":50, "C":100, "D":500, "M":1000}
```

Le code de la fonction `traduire_romain` fournie, page suivante, repose sur le principe suivant :

- la valeur d'un caractère est ajoutée à la valeur du reste de la chaîne si ce caractère a une valeur supérieure (ou égale) à celle du caractère qui le suit ;
- la valeur d'un caractère est retranchée à la valeur du reste de la chaîne si ce caractère a une valeur strictement inférieure à celle du caractère qui le suit.

Ainsi, XIV correspond au nombre 10 + 5 - 1 puisque :

- la valeur de X (10) est supérieure à celle de I (1), on ajoute donc 10 à la valeur du reste de la chaîne, c'est-à-dire IV ;
- la valeur de I (1) est strictement inférieure à celle de V (5), on soustrait donc 1 à la valeur du reste de la chaîne, c'est-à-dire V.

On rappelle que pour priver une chaîne de caractères de son premier caractère, on utilisera l'instruction :

```
nom_de_variable[1:]
```

Par exemple, si la variable `mot` contient la chaîne `"CDI"`, `mot[1:]` renvoie `"DI"`.

```
def traduire_romain(nombre):
    """ Renvoie l'écriture décimale du nombre donné en chiffres
    romains """

    if len(nombre) == 1:
        return ...
    elif romains[nombre[0]] >= ...
        return romains[nombre[0]] + ...
    else:
        return ...
```

Compléter le code de la fonction `traduire_romain` et le tester.

Exemples :

```
>>> traduire_romain("XIV")
14

>>> traduire("CXLII")
142

>>> traduire_romain("MMXXIII")
2023
```

EXERCICE 1 (4 points)

Sur le réseau social TipTop, on s'intéresse au nombre de « like » des abonnés. Les données sont stockées dans des dictionnaires où les clés sont les pseudos et les valeurs correspondantes sont les nombres de « like » comme ci-dessous :

```
{'Bob': 102, 'Ada': 201, 'Alice': 103, 'Tim': 50}
```

Écrire une fonction `max_dico` qui :

- prend en paramètre un dictionnaire `dico` non vide dont les clés sont des chaînes de caractères et les valeurs associées sont des entiers positifs ou nuls ;
- renvoie un tuple dont :
  - la première valeur est une clé du dictionnaire associée à la valeur maximale ;
  - la seconde valeur est la valeur maximale présente dans le dictionnaire.

Exemples :

```
>>> max_dico({'Bob': 102, 'Ada': 201, 'Alice': 103, 'Tim': 50})
('Ada', 201)

>>> max_dico({'Alan': 222, 'Ada': 201, 'Eve': 220, 'Tim': 50})
('Alan', 222)
```

EXERCICE 2 (4 points)

Nous avons l'habitude de noter les expressions arithmétiques avec des parenthèses comme par exemple : $(2 + 3) \times 5$.

Il existe une autre notation utilisée par certaines calculatrices, appelée notation postfixe, qui n'utilise pas de parenthèses. L'expression arithmétique précédente est alors obtenue en saisissant successivement 2, puis 3, puis l'opérateur +, puis 5, et enfin l'opérateur $\times$. On modélise cette saisie par le tableau `[2, 3, '+', 5, '*']`.

Autre exemple, la notation postfixe de $3 \times 2 + 5$ est modélisée par le tableau :

```
[3, 2, '*', 5, '+'].
```

D'une manière plus générale, la valeur associée à une expression arithmétique en notation postfixe est déterminée à l'aide d'une pile en parcourant l'expression arithmétique de gauche à droite de la façon suivante :

- si l'élément parcouru est un nombre, on le place au sommet de la pile ;
- si l'élément parcouru est un opérateur, on récupère les deux éléments situés au sommet de la pile et on leur applique l'opérateur. On place alors le résultat au sommet de la pile ;
- à la fin du parcours, il reste alors un seul élément dans la pile qui est le résultat de l'expression arithmétique.

Dans le cadre de cet exercice, on se limitera aux opérations $\times$ et + .

Pour cet exercice, on dispose d'une classe `Pile` qui implémente les méthodes de base sur la structure de pile.

Compléter le script de la fonction `eval_expression` qui reçoit en paramètre une liste python représentant la notation postfixe d'une expression arithmétique et qui renvoie sa valeur associée.

```
class Pile:
    """
    Classe définissant une structure de pile.
    """
    def __init__(self):
        self.contenu = []

    def est_vide(self):
        """
        Renvoie le booléen True si la pile est vide, False sinon.
        """
        return self.contenu == []

    def empiler(self, v):
        """
         Place l'élément v au sommet de la pile.
        """
        self.contenu.append(v)

    def depiler(self):
        """
        Retire et renvoie l'élément placé au sommet de la pile,
        si la pile n'est pas vide.
        """
        if not self.est_vide():
            return self.contenu.pop()


def eval_expression(tab):
    p = Pile()
    for ... in tab:
        if element != '+' ... element != '*':
            p.empiler(...)
        else:
            if element == ...:
                resultat = p.depiler() + ...
            else:
                resultat = ...
            p.empiler(...)
    return ...
```

Exemple :

```
>>> eval_expression([2, 3, '+', 5, '*'])
25
```

EXERCICE 1 (4 points)

Programmer la fonction `multiplication` prenant en paramètres deux nombres entiers relatifs `n1` et `n2`, et qui renvoie le produit de ces deux nombres.

Les seules opérations autorisées sont l'addition et la soustraction.

Exemples :

```
>>> multiplication(3, 5)
15
>>> multiplication(-4, -8)
32
>>> multiplication(-2, 6)
-12
>>> multiplication(-2, 0)
0
```

EXERCICE 2 (4 points)

Soit `tab` un tableau non vide d'entiers triés dans l'ordre croissant et `n` un entier.

La fonction `chercher` ci-dessous doit renvoyer un indice où la valeur `n` apparaît dans `tab` si cette valeur y figure et `None` sinon.

Les paramètres de la fonction sont :

- `tab`, le tableau dans lequel s'effectue la recherche ;
- `n`, l'entier à chercher dans le tableau ;
- `i`, l'indice de début de la partie du tableau où s'effectue la recherche ;
- `j`, l'indice de fin de la partie du tableau où s'effectue la recherche.

L'algorithme demandé est une recherche dichotomique récursive.

Recopier et compléter le code de la fonction `chercher` suivante :

```
def chercher(tab, n, i, j):
    if i < 0 or j > len(tab):
        return None
    elif i > j:
        return None
    m = (i + j) // ...
    if ... < n:
        return chercher(tab, n, ..., ...)
    elif ... > n:
        return chercher(tab, n, ..., ...)
    else:
        return ...
```

L'exécution du code doit donner :

```
>>> chercher([1, 5, 6, 6, 9, 12], 7, 0, 10)

>>> chercher([1, 5, 6, 6, 9, 12], 7, 0, 5)

>>> chercher([1, 5, 6, 6, 9, 12], 9, 0, 5)
4
>>> chercher([1, 5, 6, 6, 9, 12], 6, 0, 5)
2
```

EXERCICE 1 (4 points)

Écrire la fonction `maxliste`, prenant en paramètre un tableau non vide de nombres `tab` (de type `list`) et renvoyant le plus grand élément de ce tableau.

Exemples :

```
>>> maxliste([98, 12, 104, 23, 131, 9])
131

>>> maxliste([-27, 24, -3, 15])
24
```

EXERCICE 2 (4 points)

On dispose de chaînes de caractères contenant uniquement des parenthèses ouvrantes et fermantes.

Un parenthésage est correct si :

- le nombre de parenthèses ouvrantes de la chaîne est égal au nombre de parenthèses fermantes.
- en parcourant la chaîne de gauche à droite, le nombre de parenthèses déjà ouvertes doit être, à tout moment, supérieur ou égal au nombre de parenthèses déjà fermées.

Ainsi, `"((()())(()))"` est un parenthésage correct.
Les parenthésages `"())(()"` et `"(()))(()"` sont, eux, incorrects.

On dispose du code de la classe `Pile` suivant :

```
class Pile:
    """
    Classe définissant une pile
    """
    def __init__(self):
        self.valeurs = []

    def est_vide(self):
        """
        Renvoie True si la pile est vide, False sinon
        """
        return self.valeurs == []

    def empiler(self, c):
        """
        Place l'élément c au sommet de la pile
        """
        self.valeurs.append(c)

    def depiler(self):
        """
        Supprime l'élément placé au sommet de la pile, à
condition
        qu'elle soit non vide
        """
        if self.est_vide() == False:
            self.valeurs.pop()
```

On souhaite programmer une fonction `parenthesage` qui prend en paramètre une chaîne de caractères `ch` formée de parenthèses et renvoie `True` si la chaîne `ch` est bien parenthésée et `False` sinon.

Cette fonction utilise une pile et suit le principe suivant : en parcourant la chaîne de gauche à droite, si on trouve une parenthèse ouvrante, on l'empile au sommet de la pile et si on trouve une parenthèse fermante, on dépile (si possible) la parenthèse ouvrante stockée au sommet de la pile.
La chaîne est alors bien parenthésée si, à la fin du parcours, la pile est vide.
Elle est, par contre, mal parenthésée :

- si dans le parcours, on trouve une parenthèse fermante, alors que la pile est vide ;
- ou si, à la fin du parcours, la pile n'est pas vide.

```
def parenthesage(ch):
    """
    Renvoie True si la chaîne ch est bien parenthésée
    et False sinon
    """
    p = Pile()
    for c in ch:
        if c == ...:
            p.empiler(c)
        elif c == ...:
            if p.est_vide():
                return ...
            else:
                ...
    return p.est_vide()
```

Compléter le code de la function `parenthesage` et le tester.

Exemples :

```
>>> parenthesage("((()())(()))")
True
>>> parenthesage("())(()")
False
>>> parenthesage("(())(()")
False
```

EXERCICE 1 (4 points)

On modélise la représentation binaire d'un entier non signé par un tableau d'entiers dont les éléments sont 0 ou 1. Par exemple, le tableau `[1, 0, 1, 0, 0, 1, 1]` représente l'écriture binaire de l'entier dont l'écriture décimale est

`2**6 + 2**4 + 2**1 + 2**0 = 83`.

À l'aide d'un parcours séquentiel, écrire la fonction `convertir` répondant aux spécifications suivantes :

```
def convertir(tab):
    """
     tab est un tableau d'entiers, dont les éléments sont 0 ou 1,
     et représentant un entier écrit en binaire.
     Renvoie l'écriture décimale de l'entier positif dont la
     représentation binaire est donnée par le tableau tab
    """
```

Exemple :

```
>>> convertir([1, 0, 1, 0, 0, 1, 1])
83
>>> convertir([1, 0, 0, 0, 0, 0, 1, 0])
130
```

EXERCICE 2 (4 points)

La fonction `tri_insertion` suivante prend en argument une liste `tab` et trie cette liste en utilisant la méthode du tri par insertion. Compléter cette fonction pour qu'elle réponde à la spécification demandée.

On rappelle le principe du tri par insertion : on considère les éléments à trier un par un, le premier élément constituant, à lui tout seul, une liste triée de longueur 1. On range ensuite le second élément pour constituer une liste triée de longueur 2, puis on range le troisième élément pour avoir une liste triée de longueur 3 et ainsi de suite… A chaque étape, le premier élément de la sous-liste non triée est placé dans la sous-liste des éléments déjà triés de sorte que cette sous-liste demeure triée.
Le principe du tri par insertion est donc d'insérer à la n-ième itération, le n-ième élément à la bonne place.

```
def tri_insertion(tab):


n = len(tab)
for i in range(1, n):
    valeur_insertion = tab[...]
    # la variable j est utilisée pour déterminer où placer la
valeur à insérer
    j = ...
    # tant qu'on a pas trouvé la place de l'élément à insérer
    # on décale les valeurs du tableau vers la droite
    while j > ... and valeur_insertion < tab[...]:
        tab[j] = tab[j-1]
        j = ...
    tab[j] = ...
```

Exemple :

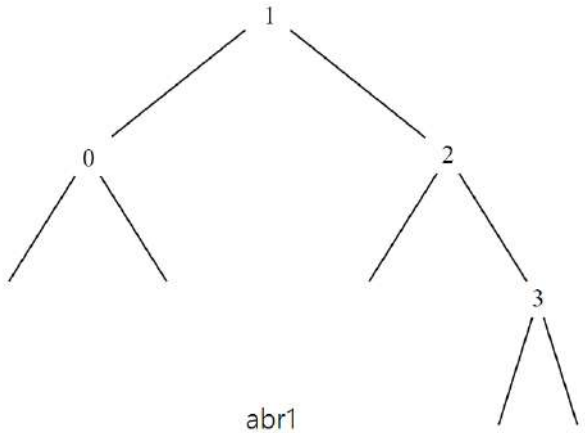
```
>>> liste = [9, 5, 8, 4, 0, 2, 7, 1, 10, 3, 6]

>>> tri_insertion(liste)

>>> liste
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

EXERCICE 1 (4 points)

On considère la classe `ABR`, dont le constructeur est le suivant :

```
def __init__(self, g0, v0, d0):
     self.gauche = g0
     self.cle = v0
     self.droit = d0
```

Ainsi, l'arbre binaire de recherche abr1 ci-contre est créé par le code python ci-dessous :




```
n0 = ABR(None, 0, None)
n3 = ABR(None, 3, None)
n2 = ABR(None, 2, n3)
abr1 = ABR(n0, 1, n2)
```

Dans tout le code, `None` correspondra à un arbre vide.

La classe `ABR` dispose aussi d'une méthode de représentation, qui affiche entre parenthèses le contenu du sous arbre gauche, puis la clé de l'arbre, et enfin le contenu du sous arbre droit. Elle s'utilise en console de la manière suivante :

```
>>>abr1
((None,0,None),1,(None,2,(None,3,None)))
```

Écrire une fonction récursive `ajoute(cle,a)` qui prend en paramètres une clé `cle` et un arbre binaire de recherche `a`, et qui renvoie un arbre binaire de recherche dans lequel `cle` a été insérée.
Dans le cas où `cle` est déjà présente dans `a`, la fonction renvoie l'arbre `a` inchangé.

Résultats à obtenir :

```
>>> a = ajoute(4, abr1)
>>> a
((None,0,None),1,(None,2,(None,3,(None,4,None))))

>>> ajoute(-5, abr1)
(((None,-5,None),0,None),1,(None,2,(None,3,None)))

>>> ajoute(2, abr1)
((None,0,None),1,(None,2,(None,3,None)))
```

EXERCICE 2 (4 points)

On dispose d'un ensemble d'objets dont on connaît, pour chacun, la masse. On souhaite ranger l'ensemble de ces objets dans des boites identiques de telle manière que la somme des masses des objets contenus dans une boîte ne dépasse pas la capacité `c` de la boîte. On souhaite utiliser le moins de boîtes possibles pour ranger cet ensemble d'objets.

Pour résoudre ce problème, on utilisera un algorithme glouton consistant à placer chacun des objets dans la première boîte où cela est possible.
Par exemple, pour ranger dans des boîtes de capacité `c = 5` un ensemble de trois objets dont les masses sont représentées en Python par la liste `[1, 5, 2]`, on procède de la façon suivante :

- Le premier objet, de masse 1, va dans une première boite.
- Le deuxième objet, de masse 5, ne peut pas aller dans la même boite que le premier objet car cela dépasserait la capacité de la boite. On place donc cet objet dans une deuxième boîte.
- Le troisième objet, de masse 2, va dans la première boîte.

On a donc utilisé deux boîtes de capacité `c = 5` pour ranger les 3 objets.

Compléter la fonction Python `empaqueter(liste_masses, c)` suivante pour qu'elle renvoie le nombre de boîtes de capacité `c` nécessaires pour empaqueter un ensemble d'objets dont les masses sont contenues dans la liste `liste_masses`.

```
def empaqueter(liste_masses, c):
    n = len(liste_masses)
    nb_boites = 0
    boites = [0]*n
    for masse in ... :
        i = 0
        while i <= nb_boites and boites[i] + ... > C:
            i = i + 1
        if i == nb_boites + 1:
            ...
    boites[i] = ...
    return ...
```

Tester ensuite votre fonction :

```
>>>empaqueter([7, 6, 3, 4, 8, 5, 9, 2], 11)
5
```

EXERCICE 1 (4 points)

Écrire en langage Python une fonction `recherche` prenant comme paramètres une variable `a` de type numérique (`float` ou `int`) et un tableau `tab` (de type `list`) et qui renvoie le nombre d'occurrences de `a` dans `tab`.

Exemples d'utilisations de la fonction `recherche` :

```
>>> recherche(5, [])
0
>>> recherche(5, [-2, 3, 4, 8])
0
>>> recherche(5, [-2, 3, 1, 5, 3, 7, 4])
1
>>> recherche(5, [-2, 5, 3, 5, 4, 5])
3
```

EXERCICE 2 (4 points)

La fonction `rendu_monnaie` prend en paramètres deux nombres entiers positifs `somme_due` et `somme_versee`. Elle procède au rendu de la monnaie de la différence `somme_versee - somme_due` pour des achats effectués avec le système monétaire de la zone Euro. On utilise pour cela un algorithme glouton qui commence par rendre le maximum de pièces ou billets de plus grandes valeurs et ainsi de suite. Par la suite, on assimilera les billets à des pièces.

La fonction `rendu_monnaie` renvoie un tableau de type `list` contenant les pièces qui composent le rendu.

Toutes les sommes sont exprimées en euros. Les valeurs possibles pour les pièces sont donc contenues dans le tableau `pieces = [1, 2, 5, 10, 20, 50, 100, 200]`.

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):
    rendu = ...
    a_rendre = ...
    i = len(pieces) - 1
    while ... :
        if pieces[i] <= a_rendre:
            rendu.append(...)
            a_rendre = ...
        else:
            i = ...
    return rendu
```

Compléter ce code et le tester.

Exemples :

```
>>> rendu_monnaie(700, 700)
[]

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

EXERCICE 1 (4 points)

Écrire une fonction `recherche` qui prend en paramètres un nombre entier `elt` et un tableau `tab` de nombres entiers, et qui renvoie l'indice de la première occurrence de `elt` dans `tab` si `elt` est dans `tab` et `-1` sinon.

Ne pas oublier d'ajouter au corps de la fonction une documentation et une ou plusieurs assertions pour vérifier les pré-conditions.

Exemples :

```
>>> recherche(1, [2, 3, 4])
-1
>>> recherche(1, [10, 12, 1, 56])
2
>>> recherche(50, [1, 50, 1])
1
>>> recherche(15, [8, 9, 10, 15])
3
>>> recherche(50, [])
-1
>>> recherche(4, [2, 4, 4, 3, 4])
1
```

EXERCICE 2 (4 points)

On considère la fonction `insere` ci-dessous qui prend en arguments un entier `a` et un tableau `tab` d'entiers triés par ordre croissant. Cette fonction crée et renvoie un nouveau tableau à partir de celui fourni en paramètre en y insérant la valeur `a` de sorte que le tableau renvoyé soit encore trié par ordre croissant. Les tableaux seront représentés sous la forme de listes Python.

```
def insere(a, tab):
    """ Insère l'élément a (int) dans le tableau tab (list)
        trié par ordre croissant à sa place et renvoie le
        nouveau tableau. """
    l = list(tab) #l contient les mêmes éléments que tab
    l.append(a)
    i = ...
    while a < ... and i >= 0:
      l[i+1] = ...
      l[i] = a
      i = ...
    return l
```

Compléter la fonction `insere` ci-dessus.

Exemples :

```
>>> insere(3, [1, 2, 4, 5])
[1, 2, 3, 4, 5]
>>> insere(30, [1, 2, 7, 12, 14, 25])
[1, 2, 7, 12, 14, 25, 30]
>>> insere(1, [2, 3, 4])
[1, 2, 3, 4]
>>> insere(1, [])
[1]
```

EXERCICE 1 (4 points)

On a relevé les valeurs moyennes annuelles des températures à Paris pour la période allant de 2013 à 2019. Les résultats ont été récupérés sous la forme de deux listes : l'une pour les températures, l'autre pour les années :

```
t_moy = [14.9, 13.3, 13.1, 12.5, 13.0, 13.6, 13.7]
annees = [2013, 2014, 2015, 2016, 2017, 2018, 2019]
```

Écrire la fonction `mini` qui prend en paramètres un tableau `releve` des relevés et un tableau `date` des dates et qui renvoie la plus petite valeur relevée au cours de la période et l'année correspondante. On suppose que la température minimale est atteinte une seule fois.

Exemple :

```
>>> mini(t_moy, annees)
(12.5, 2016)
```

EXERCICE 2 (4 points)

Un mot palindrome peut se lire de la même façon de gauche à droite ou de droite à gauche : *bob, radar,* et *non* sont des mots palindromes.

De même certains nombres sont eux aussi des palindromes : 33, *121, 345543*.

L'objectif de cet exercice est d'obtenir un programme Python permettant de tester si un nombre est un nombre palindrome.

Pour remplir cette tâche, on vous demande de compléter le code des trois fonctions ci-dessous sachant que la fonction `est_nbre_palindrome` s'appuiera sur la fonction `est_palindrome` qui elle-même s'appuiera sur la fonction `inverse_chaine`.

La fonction `inverse_chaine` inverse l'ordre des caractères d'une chaîne de caractères `chaine` et renvoie la chaîne inversée.

La fonction `est_palindrome` teste si une chaine de caractères `chaine` est un palindrome. Elle renvoie `True` si c'est le cas et `False` sinon. Cette fonction s'appuie sur la fonction précédente.

La fonction `est_nbre_palindrome` teste si un nombre `nbre` est un palindrome. Elle renvoie `True` si c'est le cas et `False` sinon. Cette fonction s'appuie sur la fonction précédente.

Compléter le code des trois fonctions ci-dessous.

```
def inverse_chaine(chaine):
    result = ...
    for caractere in chaine:
       result = ...
    return result

def est_palindrome(chaine):
    inverse = inverse_chaine(chaine)
    return ...

def est_nbre_palindrome(nbre):
    chaine = ...
    return est_palindrome(chaine)
```

Exemples :

```
>>> inverse_chaine('bac')
'cab'

>>> est_palindrome('NSI')
False

>>> est_palindrome('ISN-NSI')
True

>>>est_nbre_palindrome(214312)
False

>>>est_nbre_palindrome(213312)
True
```

EXERCICE 1 (4 points)

Écrire une fonction `recherche_indices_classement` qui prend en paramètres un entier `elt` et une liste d'entiers `tab`, et qui renvoie trois listes :

- la première liste contient les indices des valeurs de la liste `tab` strictement inférieures à `elt` ;
- la deuxième liste contient les indices des valeurs de la liste `tab` égales à `elt` ;
- la troisième liste contient les indices des valeurs de la liste `tab` strictement supérieures à `elt`.

Exemples :

```
>>> recherche_indices_classement(3, [1, 3, 4, 2, 4, 6, 3, 0])
([0, 3, 7], [1, 6], [2, 4, 5])

>>> recherche_indices_classement(3, [1, 4, 2, 4, 6, 0])
([0, 2, 5], [], [1, 3, 4])

>>>recherche_indices_classement(3, [1, 1, 1, 1])
([0, 1, 2, 3], [], [])

>>> recherche_indices_classement(3, [])
([], [], [])
```

EXERCICE 2 (4 points)

Un professeur de NSI décide de gérer les résultats de sa classe sous la forme d'un dictionnaire :

- les clefs sont les noms des élèves ;
- les valeurs sont des dictionnaires dont les clefs sont les types d'épreuves sous forme de chaîne de caractères et les valeurs sont les notes obtenues associées à leurs coefficients dans une liste.

Avec :

```
resultats = {'Dupont': {
                            'DS1': [15.5, 4],
                            'DM1': [14.5, 1],
                            'DS2': [13, 4],
                            'PROJET1': [16, 3],
                            'DS3': [14, 4]
                        },
             'Durand': {
                            'DS1': [6 , 4],
                            'DM1': [14.5, 1],
                            'DS2': [8, 4],
                            'PROJET1': [9, 3],
                            'IE1': [7, 2],
                            'DS3': [8, 4],
                            'DS4':[15, 4]
                        }
            }
```

L'élève dont le nom est Durand a ainsi obtenu au DS2 la note de 8 avec un coefficient 4**.**

Le professeur crée une fonction `moyenne` qui prend en paramètre le nom d'un de ses élèves et renvoie sa moyenne arrondie au dixième.

Compléter le code du professeur ci-dessous :

```
def moyenne(nom, dico_result):
    if nom in ...:
        notes = dico_result[nom]
        total_points = ...
        total_coefficients = ...
        for ...  in notes.values():
            note, coefficient = valeurs
            total_points = total_points + ... * coefficient
            total_coefficients = ... + coefficient
        return round( ... / total_coefficients, 1 )
    else:
        return -1
```

EXERCICE 1 (4 points)

Écrire une fonction `moyenne(liste_notes)` qui renvoie la moyenne pondérée des résultats contenus dans la liste `liste_notes`, non vide, donnée en paramètre. Cette liste contient des couples `(note, coefficient)` dans lesquels :

- `note` est un nombre de type `float` compris entre 0 et 20 ;
- `coefficient` est un nombre entier strictement positif.

Ainsi, l'expression `moyenne([(15, 2), (9, 1), (12, 3)])` devra renvoyer `12.5` :

$$\frac{15 \times 2 + 9 \times 1 + 12 \times 3}{2 + 1 + 3} = 12{,}5$$

EXERCICE 2 (4 points)

On cherche à déterminer les valeurs du triangle de Pascal (Figure 1).

Dans le triangle de Pascal, chaque ligne commence et se termine par le nombre 1. Comme l'illustre la Figure 2, on additionne deux valeurs successives d'une ligne pour obtenir la valeur qui se situe sous la deuxième valeur.

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1

Figure 1 : triangle de Pascal

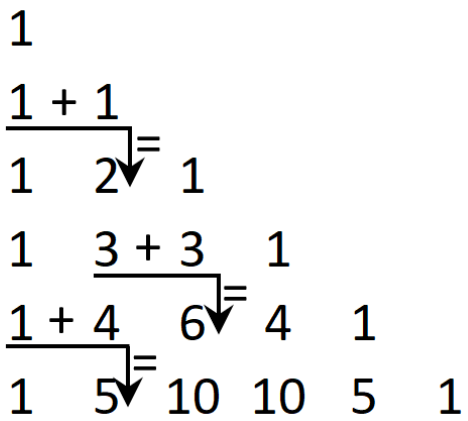


Figure 2 : méthode de calcul

Compléter la fonction `pascal` ci-après prenant en paramètre un entier `n` supérieur ou égal à 2. Cette fonction doit renvoyer une liste correspondant au triangle de Pascal de la ligne 0 à la ligne `n`. Le tableau représentant le triangle de Pascal sera contenu dans la variable `triangle`.

```
def pascal(n):
    triangle= [[1]]
    for k in range(1,...):
        ligne_k = [...]
        for i in range(1, k):
            ligne_k.append(triangle[...][i-1] + triangle[...][...])
        ligne_k.append(...)
        triangle.append(ligne_k)
    return triangle
```

Pour n = 4, voici ce que l'on devra obtenir :

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

Et pour n = 5, voici ce que l'on devra obtenir :

```
>> pascal(5)
[[1], [1, 1], [1, 2, 1], [1, 3, 3, 1], [1, 4, 6, 4, 1],
 [1, 5, 10, 10, 5, 1]]
```

EXERCICE 1 (4 points)

Écrire une fonction `max_et_indice` qui prend en paramètre une liste non vide `tab` de nombres entiers et qui renvoie la valeur du plus grand élément de cette liste ainsi que l'indice de sa première apparition dans cette liste.

L'utilisation de la fonction native `max` n'est pas autorisée.

Ne pas oublier d'ajouter au corps de la fonction une documentation et une ou plusieurs assertions pour vérifier les pré-conditions.

Exemples :

```
>>> max_et_indice([1, 5, 6, 9, 1, 2, 3, 7, 9, 8])
(9, 3)
>>> max_et_indice([-2])
(-2, 0)
>>> max_et_indice([-1, -1, 3, 3, 3])
(3, 2)
>>> max_et_indice([1, 1, 1, 1])
(1, 0)
```

EXERCICE 2 (4 points)

L'ordre des gènes sur un chromosome est représenté par un tableau `ordre` de `n` cases d'entiers distincts deux à deux et compris entre 1 et n.

Par exemple, `ordre = [5, 4, 3, 6, 7, 2, 1, 8, 9]` dans le cas `n=9`.

On dit qu'il y a un point de rupture dans `ordre` dans chacune des situations suivantes :

- la première valeur de `ordre` n'est pas 1 ;
- l'écart entre deux gènes consécutifs n'est pas égal à 1 ;
- la dernière valeur de `ordre` n'est pas n.

Par exemple, si `ordre = [5, 4, 3, 6, 7, 2, 1, 8, 9]` avec `n = 9`, on a

- un point de rupture au début car 5 est différent de 1
- un point de rupture entre 3 et 6 (l'écart est de 3)
- un point de rupture entre 7 et 2 (l'écart est de 5)
- un point de rupture entre 1 et 8 (l'écart est de 7)

Il y a donc 4 points de rupture.

Compléter les fonctions Python `est_un_ordre` et `nombre_points_rupture` proposées à la page suivante pour que :

- la fonction `est_un_ordre` renvoie `True` si le tableau passé en paramètre représente bien un ordre de gènes de chromosome et `False` sinon ;
- la fonction `nombre_points_rupture` renvoie le nombre de points de rupture d'un tableau passé en paramètre représentant l'ordre de gènes d'un chromosome.

```
def est_un_ordre(tab):
    '''
    Renvoie True si tab est de longueur n et contient tous les
entiers de 1 à n, False sinon
    '''
    for i in range(1,...):
        if ...:
            return False
    return True

def nombre_points_rupture(ordre):
    '''
    Renvoie le nombre de point de rupture de ordre qui représente
un ordre de gènes de chromosome
    '''
    assert ... # ordre n'est pas un ordre de gènes
    n = len(ordre)
    nb = 0
    if ordre[...] != 1: # le premier n'est pas 1
        nb = nb + 1
    i = 0
    while i < ...:
        if ... not in [-1, 1]: # l'écart n'est pas 1
            nb = nb + 1
        i = i + 1
    if ordre[...] != n: # le dernier n'est pas n
        nb = nb + 1
    return nb
```

Exemples :

```
>>> est_un_ordre([1, 6, 2, 8, 3, 7])
False
>>> est_un_ordre([5, 4, 3, 6, 7, 2, 1, 8, 9])
True
>>> nombre_points_rupture([5, 4, 3, 6, 7, 2, 1, 8, 9])
4
>>> nombre_points_rupture([1, 2, 3, 4, 5])
0
>>> nombre_points_rupture([1, 6, 2, 8, 3, 7, 4, 5])
7
>>> nombre_points_rupture([2, 1, 3, 4])
2
```

EXERCICE 1 (4 points)

Écrire une fonction `recherche` qui prend en paramètres un tableau `tab` de nombres entiers triés par ordre croissant et un nombre entier `n`, et qui effectue une recherche dichotomique du nombre entier `n` dans le tableau non vide `tab`.

Cette fonction doit renvoyer un indice correspondant au nombre cherché s'il est dans le tableau, `-1` sinon.

Exemples:

```
>>> recherche([2, 3, 4, 5, 6], 5)
3
>>> recherche([2, 3, 4, 6, 7], 5)
-1
```

EXERCICE 2 (4 points)

Le codage de César transforme un message en changeant chaque lettre en la décalant dans l'alphabet.

Par exemple, avec un décalage de 3, le A se transforme en D, le B en E, ..., le X en A, le Y en B et le Z en C. Les autres caractères (espace ou caractères de ponctuation : '!',' ?'…) ne sont pas codés.

La fonction `position_alphabet` ci-dessous prend en paramètre un caractère `lettre` et renvoie la position de `lettre` dans la chaîne de caractères `ALPHABET` s'il s'y trouve.

La fonction `cesar` prend en paramètre une chaîne de caractères `message` et un nombre entier `decalage` et renvoie le nouveau message codé avec le codage de César utilisant le décalage `decalage`.

```
ALPHABET = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

def position_alphabet(lettre):
    return ord(lettre) - ord('A')

def cesar(message, decalage):
    resultat = ''
    for ... in message:
        if 'A' <= c and c <= 'Z':
            indice = ( ... ) % 26
            resultat = resultat + ALPHABET[indice]
        else:
            resultat = ...
    return resultat
```

Compléter la fonction `cesar`.

Exemples :

```
>>> cesar('BONJOUR A TOUS. VIVE LA MATIERE NSI !', 4)
'FSRNSYV E XSYW. ZMZI PE QEXMIVI RWM !'


>>> cesar('GTSOTZW F YTZX. ANAJ QF RFYNJWJ SXN !', -5)
'BONJOUR A TOUS. VIVE LA MATIERE NSI !'
```

EXERCICE 1 (4 points)

Écrire une fonction `ajoute_dictionnaires` qui prend en paramètres deux dictionnaires `d1` et `d2` dont les clés sont des nombres et renvoie le dictionnaire `d` défini de la façon suivante :

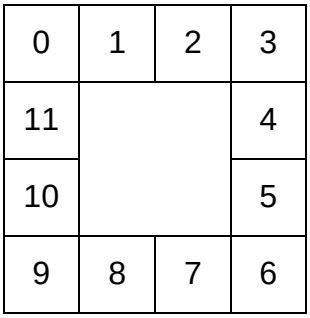
- Les clés de `d` sont celles de `d1` et celles de `d2` réunies.
- Si une clé est présente dans les deux dictionnaires `d1` et `d2`, sa valeur associée dans le dictionnaire `d` est la somme de ses valeurs dans les dictionnaires `d1` et `d2`.
- Si une clé n'est présente que dans un des deux dictionnaires, sa valeur associée dans le dictionnaire `d` est la même que sa valeur dans le dictionnaire où elle est présente.

Exemples :

```
>>> ajoute_dictionnaires({1: 5, 2: 7}, {2: 9, 3: 11})
{1: 5, 2: 16, 3: 11}

>>> ajoute_dictionnaires({}, {2: 9, 3: 11})
{2: 9, 3: 11}

>>> ajoute_dictionnaires({1: 5, 2: 7}, {})
{1: 5, 2: 7}
```

EXERCICE 2 (4 points)

On considère une piste carrée qui contient 4 cases par côté. Les cases sont numérotées de 0 inclus à 12 exclu comme ci-dessous :



L'objectif de l'exercice est d'implémenter le jeu suivant :
Au départ, le joueur place son pion sur la case 0. A chaque coup, il lance un dé équilibré à six faces et avance son pion d'autant de cases que le nombre indiqué par le dé (entre 1 et 6 inclus) dans le sens des aiguilles d'une montre.
Par exemple, s'il obtient 2 au premier lancer, il pose son pion sur la case 2 puis s'il obtient 6 au deuxième lancer, il le pose sur la case 8, puis s'il obtient à nouveau 6, il pose le pion sur la case 2.
Le jeu se termine lorsque le joueur a posé son pion **sur toutes les cases** de la piste.

Compléter la fonction `nbre_coups` ci-dessous de sorte qu'elle renvoie le nombre de lancers aléatoires nécessaires pour terminer le jeu.

Proposer ensuite quelques tests pour en vérifier le fonctionnement.

```
from random import randint

def nbre_coups():
    n = ...
    cases_vues = [0]
    case_en_cours = 0
    nbre_cases = 12
    while ... < ...:
        x = randint(1, 6)
        case_en_cours = (case_en_cours + ...) % ...
        if ...:
            cases_vues.append(case_en_cours)
        n = ...
    return n
```

EXERCICE 1 (4 points)

Le codage par différence (*delta encoding* en anglais) permet de compresser un tableau de données en indiquant pour chaque donnée, sa différence avec la précédente (plutôt que la donnée elle-même). On se retrouve alors avec un tableau de données plus petit, nécessitant donc moins de place en mémoire. Cette méthode se révèle efficace lorsque les valeurs consécutives sont proches.

Programmer la fonction `delta(liste)` qui prend en paramètre un tableau non vide de nombres entiers et qui renvoie un tableau contenant les valeurs entières compressées à l'aide de cette technique.

Exemples :

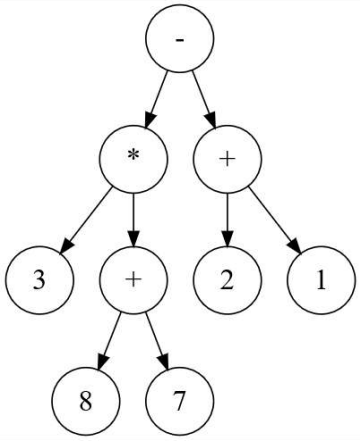
```
>>> delta([1000, 800, 802, 1000, 1003])
[1000, -200, 2, 198, 3]
>>> delta([42])
[42]
```

EXERCICE 2 (4 points)

Une expression arithmétique ne comportant que les quatre opérations $+, -, \times, \div$ peut être représentée sous forme d'arbre binaire. Les nœuds internes sont des opérateurs et les feuilles sont des nombres. Dans un tel arbre, la disposition des nœuds joue le rôle des parenthèses que nous connaissons bien.

En parcourant en profondeur infixe l'arbre binaire ci-contre, on retrouve l'expression notée habituellement :

$\big(3 \times (8 + 7)\big) - (2 + 1)$.



La classe `Noeud` ci-après permet d'implémenter une structure d'arbre binaire.

Compléter la fonction récursive `expression_infixe` qui prend en paramètre un objet de la classe `Noeud` et qui renvoie l'expression arithmétique représentée par l'arbre binaire passé en paramètre, sous forme d'une chaîne de caractères contenant des parenthèses.

Résultat attendu avec l'arbre ci-dessus :

```
>>> e = Noeud(Noeud(Noeud(None, 3, None),
       '*', Noeud(Noeud(None, 8, None), '+', Noeud(None, 7, None))),
       '-', Noeud(Noeud(None, 2, None), '+', Noeud(None, 1, None)))

>>> expression_infixe(e)
'((3*(8+7))-(2+1))'
```

```
class Noeud:
    '''
    classe implémentant un noeud d'arbre binaire
    '''

    def __init__(self, g, v, d):
        '''
        un objet Noeud possède 3 attributs :
        - gauche : le sous-arbre gauche,
        - valeur : la valeur de l'étiquette,
        - droit : le sous-arbre droit.
        '''
        self.gauche = g
        self.valeur = v
        self.droit = d

    def __str__(self):
        '''
        renvoie la représentation du noeud en chaine de
caractères
        '''
        return str(self.valeur)

    def est_une_feuille(self):
        '''
        renvoie True si et seulement si le noeud est une feuille
        '''
        return self.gauche is None and self.droit is None


def expression_infixe(e):
    s = ...
    if e.gauche is not None:
        s = '(' + s + expression_infixe(...)
    s = s + ...
    if ... is not None:
        s = s + ... + ...
    return s
```