Exercices pour l'épreuve pratique de la spécialité NSI. Série 1

Exercice 1.1

Écrire une fonction `recherche` qui prend en paramètres `caractere`, un caractère, et `mot`, une chaîne de caractères, et qui renvoie le nombre d'occurrences de `caractere` dans `mot`, c'est-à-dire le nombre de fois où `caractere` apparaît dans `mot`.
Exemples :

```
>>> recherche('e', "sciences")
2
>>> recherche('i',"mississippi")
4
>>> recherche('a',"mississippi")
0
```

```python
def recherche(caractere, mot):
    occurences = 0
    for lettre in mot:
        if lettre == caractere:
            occurences +=1
    return occurences



print(recherche('e', "sciences"))              # 2
print(recherche('i',"mississippi"))            # 4
print(recherche('a',"mississippi"))            # 0
```

Exercice 1.2

On s'intéresse à un algorithme récursif qui permet de rendre la monnaie à partir d'une liste donnée de valeurs de pièces et de billets - le système monétaire est donné sous forme d'une liste `pieces=[100, 50, 20, 10, 5, 2, 1]` - (on supposera qu'il n'y a pas de limitation quant à leur nombre), on cherche à donner la liste de pièces à rendre pour une somme donnée en argument.
Compléter le code Python ci-dessous de la fonction `rendu_glouton` qui implémente cet algorithme et renvoie la liste des pièces à rendre

```
Pieces = [100,50,20,10,5,2,1]
def rendu_glouton(arendre, solution=[], i=0):
    if arendre == 0:
        return ...
    p = pieces[i]
    if p <= ... :
        solution.append(...)
        return rendu_glouton(arendre - p, solution, i)
    else :
        return rendu_glouton(arendre, solution, ...)
```

On devra obtenir :

```
>>>rendu_glouton_r(68,[],0)
[50, 10, 5, 2, 1]
>>>rendu_glouton_r(291,[],0)
[100, 100, 50, 20, 20, 1]
```

```python
pieces = [100,50,20,10,5,2,1]
def rendu_glouton(arendre, solution=[], i=0):
    if arendre == 0:
        return solution
    p = pieces[i]
    if p <= arendre :
        solution.append(p)
        return rendu_glouton(arendre - p, solution, i)
    else :
        return rendu_glouton(arendre, solution, i+1)


print(rendu_glouton(68,[],0))              # [50, 10, 5, 2, 1]
print(rendu_glouton(291,[],0))             # [100, 100, 50, 20, 20, 1]
```

Exercice 3.1

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 assez petites nécessitant moins de place en mémoire. Cette méthode se révèle efficace lorsque les valeurs consécutives sont proches.
Programmer la fonction `delta` qui prend en paramètre un tableau non vide de nombres entiers et qui renvoie un tableau contenant les valeurs entières compressées à l'aide cette technique.

Exemples :

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

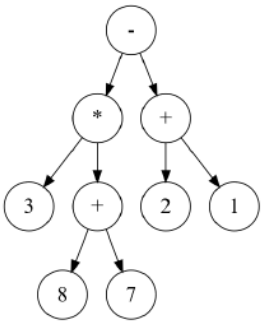
```python
def delta(tab):
    comp = [tab[0]]
    for i in range(1,len(tab)):
        comp.append(tab[i]-tab[i-1])
    return comp


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

Exercice 3.2

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 :

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



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

    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 + ...
    if ...:
        return s

    return '('+ s +')'
```

```
class Noeud:
    def __init__(self, g, v, d):
        self.gauche = g
        self.valeur = v
        self.droit = d

    def __str__(self):
        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(e.gauche)
    s = s + str(e.valeur)
    if e.droit is not None:
        s = s + expression_infixe(e.droit)
    if e.est_une_feuille():
        return s

    return '('+ s +')'


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)))

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

Exercice 4.1 : Écrire une fonction `recherche` qui prend en paramètre un tableau de nombres entiers `tab`, et qui renvoie la liste (éventuellement vide) des couples d'entiers consécutifs successifs qu'il peut y avoir dans `tab`.

Exemples :

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

```python
def recherche(tab):
    liste =[]
    for i in range(1,len(tab)):
        if tab[i] == tab[i-1] + 1:
            liste.append((tab[i-1], tab[i]))
    return liste


print(recherche([1, 4, 3, 5]))                # []
print(recherche([1, 4, 5, 3]))                # [(4, 5)]
print(recherche([7, 1, 2, 5, 3, 4]))          # [(1, 2), (3, 4)]
print(recherche([5, 1, 2, 3, 8, -5, -4, 7]))  # [(1, 2), (2, 3), (-5, -4)]
```

Exercice 5.1 : Écrire une fonction `RechercheMinMax` qui prend en paramètre un tableau de nombres non triés `tab`, et qui renvoie la plus petite et la plus grande valeur du tableau sous la forme d'un dictionnaire à deux clés 'min' et 'max'. Les tableaux seront représentés sous forme de liste Python.

Exemples :

```
>>> tableau = [0, 1, 4, 2, -2, 9, 3, 1, 7, 1] >>> resultat =
rechercheMinMax(tableau) >>> resultat
{'min': -2, 'max': 9}
>>> tableau = [] >>> resultat = rechercheMinMax(tableau) >>> resultat
{'min': None, 'max': None}
```

```python
def rechercheMinMax(tab):
    if tab == []:
        return {'min':None, 'max':None}
    minmax = {'min':tab[0], 'max':tab[0]}
    for nb in tab:
        if nb > minmax['max']:
            minmax['max'] = nb
        if nb < minmax['min']:
            minmax['min'] = nb
    return minmax




tableau = [0, 1, 4, 2, -2, 9, 3, 1, 7, 1]
resultat = rechercheMinMax(tableau)
print(resultat)                               # {'min': -2, 'max': 9}

tableau = []
resultat = rechercheMinMax(tableau)
print(resultat)                               # {'min': None, 'max': None}
```

Exercice 6.1 : Écrire une fonction `maxi` qui prend en paramètre une liste `tab` de nombres entiers et qui renvoie un couple donnant le plus grand élément de cette liste ainsi que l'indice de la première apparition de ce maximum dans la liste.

Exemple :

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

```python
def maxi(tab):
    if tab == []:
        return None
    max = tab[0]
    i_max = 0
    for i in range(1,len(tab)):
        if tab[i] > max:
            max = tab[i]
            i_max = i
    return (max, i_max)

print(maxi([1,5,6,9,1,2,3,7,9,8])) # (9,3)
```

Exercice 6.2

La fonction `recherche` prend en paramètres deux chaines de caractères `gene` et `seq_adn` et renvoie `True` si on retrouve `gene` dans `seq_adn` et `False` sinon.
Compléter le code Python ci-dessous pour qu'il implémente la fonction `recherche`.

```
def recherche(gene, seq_adn):
    n = len(seq_adn)
    g = len(gene)
    i = ...
    trouve = False
    while i < ... and trouve == ... :
        j = 0
        while j < g and gene[j] == seq_adn[i+j]:
            ...
        if j == g:
            trouve = True
        ...
    return trouve
```

Exemples :

```
>>> recherche("AATC", "GTACAAATCTTGCC")
True
>>> recherche("AGTC", "GTACAAATCTTGCC")
False
```

```python
def recherche(gene, seq_adn):
    n = len(seq_adn)
    g = len(gene)
    i = 0
    trouve = False
    while i < n and trouve == False :
        j = 0
        while j < g and gene[j] == seq_adn[i+j]:
            j += 1
        if j == g:
            trouve = True
        i += 1
    return trouve


print(recherche("AATC", "GTACAAATCTTGCC"))   # True
print(recherche("AGTC", "GTACAAATCTTGCC"))   # False
```

Exercice 7.1 :

Écrire une fonction `conv_bin` qui prend en paramètre un entier positif `n` et renvoie un couple `(b,bit)` où :

- b est une liste d'entiers correspondant à la représentation binaire de n;
- bit correspond aux nombre de bits qui constituent b.

Exemple :

```
>>> conv_bin(9)
([1,0,0,1],4)
```

Aide :

- l'opérateur `//` donne le quotient de la division euclidienne : `5//2` donne `2` ;
- l'opérateur `%` donne le reste de la division euclidienne : `5%2` donne `1` ;
- `append` est une méthode qui ajoute un élément à une liste existante :

Soit `T=[5,2,4]`, alors `T.append(10)` ajoute 10 à la liste `T`. Ainsi, `T` devient `[5,2,4,10]`.

- `reverse` est une méthode qui renverse les éléments d'une liste.

Soit `T=[5,2,4,10]`. Après `T.reverse()`, la liste devient `[10,4,2,5]`.

On remarquera qu'on récupère la représentation binaire d'un entier `n` en partant de la gauche en appliquant successivement les instructions :

```
b = n%2
n = n//2
```

répétées autant que nécessaire.

```python
def conv_bin(n):
    b = [n%2]
    n = n // 2
    while n > 0 :
        b.append(n%2)
        n = n// 2
    b.reverse()
    return (b, len(b))

print(conv_bin(12))          # ([1,1,0,0],4)
```

Exercice 7.2 :

La fonction `tri_bulles` prend en paramètre une liste `T` d'entiers non triés et renvoie la liste triée par ordre croissant.

Compléter le code Python ci-dessous qui implémente la fonction `tri_bulles`.

```
def tri_bulles(T):
    n = len(T)
    for i in range(...,...,-1):
        for j in range(i):
            if T[j] > T[...]:
                ... = T[j]
                T[j] = T[...]
                T[j+1] = temp
    return T
```

```python
from random import randint
def tri_bulles(T):
    n = len(T)
    for i in range(n-1,0,-1):
        for j in range(i):
            if T[j] > T[j+1]:
                temp = T[j]
                T[j] = T[j+1]
                T[j+1] = temp
    return T

T = [randint(1,99) for i in range(10)]
T = tri_bulles(T)
print(T)
```