TP 2

MANIPULATION DE LISTES ET DE PILES

```
## Exercice 1

# Question 1

def disjoints(i1, i2):
    """ entrée : deux intervalles i.e. deux listes [a, b] de flottants
        sortie : booléen indiquant si i1 et i2 sont disjoints
    """
    min1, max1 = i1
    min2, max2 = i2
    return max2 < min1 or max1 < min2

assert disjoints([1,2],[3,7])
assert not disjoints([1,8],[3,7])
assert not disjoints([1,8],[6,10])


# Question 2

def fusion(i1, i2):
    """ entrée : deux intervalles i1, i2 i.e. deux listes [a, b] de flottants
        sortie : intervalle fusion de i1 et i2
    """
    min1, max1 = i1
    min2, max2 = i2
    if min1 < min2 :
        mini = min1
    else:
        mini = min2
    if max1 < max2 :
        maxi = max2
    else:
        maxi = max1
    return [mini,maxi]

assert fusion([1, 2], [3, 7]) == [1, 7]
assert fusion([1, 8], [3, 7]) == [1, 8]
assert fusion([1, 3], [2, 5]) == [1, 5]


# Question 3

def verifie_iter(L):
    """ entrée : liste d'intervalles i.e. de listes [a, b] de flottants
        sortie : booléen indiquant si la liste est "bien formée"
    """
    ind = 0
    while ind < len(L)-1:
        if L[ind][1] >= L[ind+1][0]:
            return False
        ind += 1
    return True
```

```
def verifie(L):
    """ entrée : liste d'intervalles i.e. de listes [a, b] de flottants
        sortie : booléen indiquant si la liste est "bien formée"
    """
    if len(L) == 1 :
        return True
    else:
        i1 = L[0]
        reste = L[1:]
        return i1[1] < reste[0][0] and verifie(reste)


assert not verifie_iter([[0,1],[2,5],[3,6]])
assert not verifie_iter([[2,5],[0,1],[3,6]])
assert verifie_iter([[0,1],[2,3],[4,6]])

assert not verifie([[0,1],[2,5],[3,6]])
assert not verifie([[2,5],[0,1],[3,6]])
assert verifie([[0,1],[2,3],[4,6]])


# Question 4

def appartient(x, L):
    """ entrée : flottant x
                 liste L d'intervalles i.e. de listes [a, b] de flottants
        précondition : L est bien formée
        sortie : booléen indiquant si x est un élément de l'un des intervalles de L
    """
    i1 = L[0]
    if len(L) == 1 :
        return i1[0] <= x <= i1[1]
    else:
        if x < i1[0]:
            return False
        elif x <= i1[1]:
            return True
        else:
            return appartient(x,L[1:])


assert not appartient(-1,[[0,1],[2,3],[4,6]])
assert appartient(0,[[0,1],[2,3],[4,6]])
assert not appartient(3.5,[[0,1],[2,3],[4,6]])
assert appartient(5,[[0,1],[2,3],[4,6]])
assert appartient(6,[[0,1],[2,3],[4,6]])
assert not appartient(7,[[0,1],[2,3],[4,6]])



## Exercice 2

# Question 1

def estCroissante(L):
    """ entrée : liste L d'entiers
        sortie : booléen indiquant si L est croissante """
    if L == []:
        return True
    eprec = L[0]
    for e in L :
        if eprec > e:
            return False
        eprec = e
    return True

assert estCroissante([0,1,1,1,3,7])
assert not estCroissante([0,1,0,1,3,7])
```

```python
def estDecroissante(L):
    """ entrée : liste L d'entiers
        sortie : booléen indiquant si L est décroissante """
    if L == []:
        return True
    eprec = L[0]
    for e in L :
        if eprec < e:
            return False
        eprec = e
    return True

assert estDecroissante([4,2,1])

def estMonotone(L):
    """ entrée : liste L d'entiers
        sortie : booléen indiquant si L est monotone """
    return estCroissante(L) or estDecroissante(L)


# Question 2

def maxCroissante(L):
    """ entrée : liste L d'entiers
        sortie : (première) plus grande tranche croissante de L """
    Lcour = [L[0]]  # sous-liste croissante en cours
    Lmax = []       # candidat pour la plus longue liste croissante
    i = 1
    while i < len(L):
        if L[i] >= L[i-1]:  # si cela est toujours croissant...
            Lcour.append(L[i])
        else:               # sinon on arrête la liste en cours
            if len(Lcour) > len(Lmax): # et on regarde si c'est la plus longue
                Lmax = Lcour
            Lcour = [L[i]]  # on commence une nouvelle liste
        i += 1
    if len(Lcour) > len(Lmax): # il reste à traiter la dernière liste en cours
        Lmax = Lcour
    return Lmax

assert maxCroissante([0, 1, 1, 3, 7]) == [0, 1, 1, 3, 7]
assert maxCroissante([1, 1, 0, 1, 1, 1, 3, 7, 6]) == [0, 1, 1, 1, 3, 7]
assert maxCroissante(list(range(0,-4,-1))) == [0]


# Question 3a
# La liste [0, 1, 0, 1, 0] ne présente que des monotonies banales.


# Question 3b

def cahots(L):
    """ entrée : liste L d'entiers
        sortie : booléen indiquant si L ne comporte que des monotonies banales """
    if len(L) < 2:
        return False
    if L[0] == L[1]:
        return False
    croissante = L[0] < L[1]
    eprec = L[1]
    for e in L[2:]:
        if eprec == e:
            return False
        if eprec < e :
            if croissante:
                return False
        else :
            if not croissante:
                return False
        croissante = not croissante
        eprec = e
    return True
```

```
assert cahots([0,1])
assert cahots([0,2,1,3,2,3,0,1,0])
assert not cahots([1,2,3,2,1])


# Question 3c
""" Il suffit d'échanger les termes 0 et 1, les termes 2 et 3, les termes 4 et 5, etc. """




## Exercice 3 - avec le fichier Pilefile

# def sommet(p):
#     """ entrée : p pile
#         sortie : sommet de p """
#     if not p.empty():
#         a = p.depiler()
#         p.empiler(a)
#         return a
#
#
# def echange(p):
#     """ entrée : p pile
#         sortie : échange des deux éléments au sommet de p;
#                  ne fait rien si p a au plus un élément """
#     if not p.empty():
#         a = p.depiler()
#         if not p.empty():
#             b = p.depiler()
#             p.empiler(a)
#             p.empiler(b)
#         else:
#             p.empiler(a)


def bas1(p):
    """ entrée : p pile
        sortie : renvoie le bas de p sans changer p """
    q = Pile()
    while not p.empty():
        a = p.depiler()
        q.empiler(a)
    return a


def bas2(p):
    """ entrée : p pile
        sortie : renvoie le bas de p sans changer p """
    q = Pile()
    while not p.empty():
        a = p.depiler()
        q.empiler(a)
    while not q.empty():
        p.empiler(q.depiler())
    return a


def inside(p, x):
    """ entrée : p pile, x objet
        sortie : booléen indiquant si x apparaît dans p """
    test = False
    q = Pile()
    while not p.empty() and test == False:
        a = p.depiler()
        q.empiler(a)
        if a == x:
            test = True
    while not q.empty():
        p.empiler(q.depiler())
    return test
```

```
# def taille(p):
#     """ entrée : p pile
#         sortie : taille de p (entier), sans détruire p """
#     n = 0
#     q = Pile()
#     while not p.empty():
#         q.empiler(p.depiler())
#         n += 1
#     while not q.empty():
#         p.empiler(q.depiler())
#     return n

def renverse(p):
    """ entrée : p pile
        renverse la pile p
        sortie : None """
    q = Pile()
    r = Pile()
    while not p.empty():
        q.empiler(p.depiler())
    while not q.empty():
        r.empiler(q.depiler())
    while not r.empty():
        p.empiler(r.depiler())



def copie(p):
    """ entrée : p pile
        sortie : renvoie une copie de p et ne modifie pas p """
    q = Pile()
    r = Pile()
    while not p.empty():
        q.empiler(p.depiler())
    while not q.empty():
        a = q.depiler()
        p.empiler(a)
        r.empiler(a)
    return r
```