TP 1

MANIPULATION DE LISTES

```
## Exercice 1

# Question 1

def cardinal(A):
    c = 0
    for el in A:
        c += el
    return(c)


# Question 2

def intersection(A, B):
    inters = []
    for k in range(len(A)):
        if A[k]==1 and B[k]==1:
            inters.append(1)
        else:
            inters.append(0)
    return(inters)


# Question 3

def difference(A, B):
    dif = []
    for k in range(len(A)):
        if A[k] == 1 and B[k] == 0:
            dif.append(1)
        else:
            dif.append(0)
    return dif


# Question 4.a

def decoder(E,C):
    P = []
    for k in range(len(E)):
        if C[k] == 1:
            P.append(E[k])
    return P
```

```python
# Question 4.b

def coder(E, P):
    C = [0 for k in range(len(E))]
    for i in range(len(P)):
        for j in range(len(E)):  # on cherche où est l'élément dans E
            if P[i] == E[j]:
                C[j] = 1
    return C


# Question 4.c

def incrementer(C):
    D = [x for x in C]  # ou D = C.copy()
    no = len(D)-1
    while D[no] == 1 :
        D[no] = 0
        no -= 1
    if no != -1:
        D[no] = 1
    return D


# Question 4.d

def parties(E):
    C = [0]*len(E)
    LP = [[]]
    for no in range(1,2**(len(E))) :
        C = incrementer(C)
        LP.append(decoder(E,C))
    return LP


# Question 4.e
def p_parties1(E,p):
    if p == 0:
        return []
    n = len(E)
    C = n*[0]
    LP = []
    for no in range(1,2**n) :
        C = incrementer(C)
        if cardinal(C)==p:
            LP.append(decoder(E,C))
    return LP

for i in range(5) :
    print(p_parties1([2,3,5,7],i))


# Question 5.a

def reunion(C1, C2):
    C = [x for x in C1]
    for k in range(len(C)):
        if C2[k] == 1:
            C[k] = 1
    return C


# Question 5.b

def estRec(R) :
    codeU = R[0]
    for C in R[1:] :
        codeU = reunion(codeU, C)
    return codeU == len(codeU)*[1]
```

```python
# Question 5.c
''' Il suffit de vérifier que si l'on enlève une partie,
    alors R n'est plus un recouvrement.'''

def estRecMin(R) :
    if not estRec(R) :
        return False
    for i in range(len(R)) :
        Rprime = R[:i] + R[i+1:]
        if estRec(Rprime):
            return False
    return True


# Question 5.d
''' Il suffit de vérifier, lorsque R est un recouvrement,
    que la somme des cardinaux des éléments de R vaut n
    et que les éléments de R sont non vides.'''

def estPartition(R) :
    if not estRec(R) :
        return False
    cardinaux = [sum(A) for A in R]
    if 0 in cardinaux :  # il y a une partie vide
        return False
    return sum(cardinaux) == len(R[0])


# Question 5.e
def partition(R) :
    n = len(R[0])
    reste = [1 for i in range(n)]
    P = []
    for e in R :
        new = True
        if e == [0 for i in range(n)]:
            new = False
        else:
            for i in range(n):
                if e[i] == 1 and reste[i] == 0:
                    new = False
        if new:
            for i in range(n):
                if e[i] == 1:
                    reste[i] = 0
            P.append(e)
    if sum(reste) != 0:
        P.append(reste)
    return P

assert partition([[0,0,0]]) == [[1,1,1]]
assert partition([[1,1,0],[1,1,1]]) == [[1,1,0],[0,0,1]]
assert partition([[1,0,0],[0,1,0]]) == [[1,0,0],[0,1,0],[0,0,1]]
assert partition([[1,1,0,0],[1,0,0,0],[0,0,0,1]]) == [[1,1,0,0],[0,0,0,1],[0,0,1,0]]




## Exercice 2

# 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])
```

```python
# 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, L liste 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 3

# 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])


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 """
    if L == []:
        return []
    nmax = n = 1
    SL = r = [L[0]]
    for e in L[1:]:
        if r[-1] <= e : # sous-liste croissante en cours...
            n += 1
            r.append(e)
        else: # sous-liste terminée
            if n > nmax :
                nmax = n
                SL = r
            r = [e]
            n = 1
    if n > nmax:
        nmax = n
        SL = r
    return SL
```

```
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. """

from random import randint
L = [randint(-10,10) for n in range(15)]
L.sort()
print("Liste tirée :",L)

newL = []
n = len(L)

if n%2 == 0 : # nombre pair de termes
    for i in range(n//2):
        newL.append(L[:n//2][i])
        newL.append(L[n//2:][i])
else: # nombre impair de termes
    for i in range((n-1)//2):
        newL.append(L[:(n-1)//2][i])
        newL.append(L[(n-1)//2:][i])
    newL.append(L[(n-1)//2])

print("Liste réordonnée :",newL)
print("cahots(liste) ->",cahots(newL))
```