```python
# Correction_DS4_Info_MPSI_2023_2024.py

##Exercice. Représentation de Zeckendorf

#Q1.
'''F0 = 1, F1 = 2, F2 = 3, F3 = 5, F4 = 8, F5 = 13,
F6 = 21, F7 = 34, F8 = 55, F9 = 89'''

#Q2.
def fibo(k):
    assert k > 0  #déclenchera une erreur si k<=0
    if k == 1: return [1]
    else :
        L = [1, 2]  #initialisation à [F0,F1]
        for _ in range(k-2): #k-1 itérations (pour k = 1 :
aucune itération)
            fk2 = L[-1]+L[-2]
            L.append(fk2)
    return L

#Q3. Q4.
'''a=60 a comme décomposition : 55 + 5 soit F8 + F3
ce qui donnera la représentation de Zeckendorf :
'000100001' '''

#Q5.
def zeckToInt(s):
    k = len(s)
    L = fibo(k)  #[F0,...,Fk-1]
    a = 0      #initialisation de l'entier a cherché
    for i in range(k):
        if s[i] == '1':
            a += L[i]
    return a

#Q6a.
'''On cherche k tel que F[k] <= a < F[k+1]. On a alors l =
k+1.
Pour assurer une complexité en O(l) on ne peut pas
utiliser la fonction fibo.'''
def tailleZeck(a):
    assert a >= 1
    k = 0
    fk = 1  #fk
    fk1 = 2  #fk+1
    while a >= fk1:
```



```python
        fk2 = fk + fk1
        fk, fk1 = fk1, fk2
        k += 1
    return k+1

#Q6b.
def intToZeck(a):
    l = tailleZeck(a)
    L = fibo(l) #liste [f0,F1,...,Fk] où k est tel que
F[k] <= a < F[k+1]
    #On initialise au 1 qui se trouve à droite.
    #On va remplir la chaine de la droite vers la gauche
    zeck = '1'
    i = l-1   #indice dans la liste L
    reste = a - L[i]
    i -= 1
    while i >= 0:
        fi = L[i]
        #on va comparer ce qui reste avec fi
        #si reste >= fi et si on n'a pas pris fi+1,
        #on prend fi et on met à jour reste
        #sinon, on ne le prend pas
        if reste >= fi and zeck[0] != '1':
            reste -= fi
            zeck = '1' + zeck
        else:
            zeck = '0' + zeck
        i -= 1
    return zeck

print(intToZeck(40))
'''10010001'''

#7a.
def AtoCF(A):
    CF = ''
    for a in A:  #a décrit les éléments de A
        CF += intToZeck(a)
        CF += '1'
    return CF

print(AtoCF([5,40,21]))

#Q7b.
'''
La fin d'une représentation des différents entiers
```

```
présents dans A est
marquée par la présence de deux '1' successifs.
On utilise deux variables debut et fin pour stocker
l'indice de début et
l'indice de fin (inclus) de l'entier que l'on est en train
de décoder.
'''
def CFtoA(CF):
    A = []
    deb = 0
    fin = 0
    while fin < len(CF)-1:  #on s'arrête à l'avant
dernière valeur
        if CF[fin] == '1' and CF[fin + 1] == '1':
            #on est arrivé à la fin d'un entier
            #on décode l'entier, on le stocke, on met à
jour deb et fin
            a = zeckToInt(CF[deb:fin+1])
            A.append(a)
            deb = fin + 2
            fin = fin + 2
        else:
            #on avance normalement dans l'entier en cours
de décodage
            fin += 1
    return A

print(CFtoA('000111001000110000011'))
'''[5, 40, 21]'''

#Q8.
'''
si on note A la partie de IN telle que a = sum(fi, i in
A), alors
x**a est le produit des x**fi pour i dans A.
Or, x**(fk+2) = x**(fk+1) * x**(fk),
on calcule les x**fi de proche en proche grâce à cette
formule
en effectuant simplement des produits
'''
def puiss(x,s):
    rep = 1
    xpuis_fk = x  #x puissance fk
    xpuis_fk1 = x*x  #x puissance fk+1
    #initialisation de s en fonction des 2 premières
valeurs
```



```
    if s[0] == '1':
        rep *= xpuis_fk
    if len(s) >= 1 and s[1] == '1':
        rep *= xpuis_fk1
    while k < len(s)-2:
        xpuis_fk2 = xpuis_fk * xpuis_fk1 #x puissance fk+2
        if s[k+2] == '1':
            rep *= xpuis_fk
        k += 1  #on avance dans s et on met à jour fk et
fk+1
        xpuis_fk, xpuis_fk1 = xpuis_fk1, xpuis_fk2
    return rep

##Problème. Jeu de taquin


def copy(G):
    return [L[:] for L in G]

n = 4

Gfig1 = [[1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,0]]
Gfig2 = [[5,11,1,2],[14,10,4,7],[3,13,0,12],[9,8,6,15]]
Gfig6 = [[2,1,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,0]]

#Q1.
def test(G):
    if len(G) != n:  #n sous-listes
        return False
    for L in G:
        if len(L) != n:   #chaque sous-liste de taille n
            return False
        for x in L:
            if not (0 <= x <= n**2-1):  #tous les éléments
dans 0,n**2-1
                return False
    return True

#Q2.
def coord(G,x):
    i = 0 #indice de ligne
    while i<n:
        j = 0  #indice de colonne
        while j<n:
            if G[i][j] == x:
                return (i,j)
```

```
            j += 1
        i += 1
    return None  #cas où x n'est pas présent

#print(coord(Gfig1,16))

#Q3.
def tousPresents(G):
    n = len(G)
    for x in range(n*n):
        if coord(G,x) == None:
            return False
    return True

'''
La signature de la fonction tousPresents est :
tousPresents(G: list[list[int]]) -> bool

Complexité :
Coord(G,x) est en O(n**2).
Coord(G,x) est appelé dans la boucle for x in range(n**2).
La complexité globale est donc en O(n***4)'''

#Q4.
'''Pour obtenir une fonction en complexité quadratique :
- On crée une liste presents contenant n**2 False.
- On parcourt tous les éléments de G (avec deux boucles
imbriquées) et on remplace presents[x] par True.
(complexité O(n**2))
- Ensuite on parcourt la liste presents et on renvoie
False si l'un des éléments est False. Une fois le parcours
terminé on renvoie True. (Complexité en O(len(presents))
soit O(n**2).'''

#Q5a.
'''H est possible ssi i0 < n-1, B est possible ssi i0 > 0,
G est possible ssi j0 < n-1, D est possible ssi j0 > 0'''
def EstLicite(i0, j0, n, M):
    if M == 'H': return i0<n-1
    elif M == 'B': return i0>0
    elif M == 'G': return j0<n-1
    elif M == 'D': return j0>0
    else: return False

#Q5b.
def mvt(G1, M):
```



```
    i0,j0 = coord(G1,0) #coordonnées de la case vide
    assert EstLicite(i0, j0, n, M)
    G2 = copy(G1)
    if M == 'H':
        ix, jx = i0 + 1, j0  #ix et jx sont les
coordonnées de la case qui va être déplacée dans la case
vide
    elif M == 'B':
        ix, jx = i0 - 1, j0
    elif M == 'D':
        ix, jx = i0, j0 - 1
    elif M == 'G':
        ix, jx = i0, j0 + 1
    G2[i0][j0] = G1[ix][jx]
    G2[ix][jx] = 0
    return G2

#Q5c.
"""
- Complexité de la ligne coord(G1,0) : O(n**2)
- Complexité de la copie : O(1)
- Tests et définitions de ix,jx : O(1)
- Modification de deux valeurs de G2 : O(1)
Au global : O(n**2)"""

#Q6a.
"""à la fin des 3 tours de la 1ère boucle for, GF vaut :
[[1,2,3],[4,5,6],[7,8,9]]
à la fin de la fonction, GF vaut : [[1,2,3],[4,5,6],
[7,8,0]]"""

#Q6b.
def f(n):
    return [[0 for _ in range(n)] for _ in range(n)]

def g(i,j,n):
    if i == n-1 and j == n-1: return 0
    else: return j + 1 + i*n

#Q7.
def verif(G1, G2, LM):
    G = copy(G1)  #Grille initiale
    for mouvement in LM:
        G = mvt(G, mouvement)
    return G == G2
```

```python
#Q8a.
def nbCarac(G):
    i0, j0 = coord(G,0)
    LG = []
    for L in G:
        LG += L
    d = 0
    for k1 in range(len(LG)):
        for k2 in range(k1+1,len(LG)): #assure k2 > k1
            if LG[k1] >LG[k2]:
                d += 1
    return d + i0 + j0

#Q8b.
'''
Complexité de la fonction Coord : O(n**2)
Complexité de la construction de LG : O(n)
Complexité des deux boucles for imbriquées : sum(n**2-
k1,k1=0..n**2) soit O (n**4)
Au global : O(n**2)+O(n)+O(n**4) = O(n**4)'''

#Q9a.
'''Pour la figure 1 : LG =
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0]
i0 = j0 = 3, d = 15 donc c1 = 21
Pour la figure 6 : LG =
[2,1,3,4,5,6,7,8,9,10,11,12,13,14,15,0]
i0 = j0 = 3, d = 16 (couple k1 = 0, k2 = 1) donc c6 = 23
c1 et c6 n'ont pas la même parité donc la figure 6 n'est
pas soluble'''

#Q9b.
def testParite(G1, G2):
    if (nbCarac(G1) - nbCarac(G2))%2 ==0:
        return True
    return False

#Q10.
def deplVide(G, x):
    i0, j0 = coord(G, 0)
    G1 = copy(G)
    #déplacement de la case vide en bas à droite
    LM = []
    while EstLicite(i0, j0, n, 'H'):
        G1 = mvt(G1, 'H')
        i0, j0 = coord(G1, 0)
```



```python
        LM.append('H')
    while EstLicite(i0, j0, n, 'G'):
        G1 = mvt(G1, 'G')
        i0, j0 = coord(G1, 0)
        LM.append('G')
    #déplacement de la case vide à côté de x
    i,j = coord(G1, x)
    if i == n-1:
        while j0 > j+1:
            G1 = mvt(G1, 'D')
            LM.append('D')
            j0 -= 1
    else:
        while j0 > j:
            G1 = mvt(G1, 'D')
            LM.append('D')
            j0 -= 1
        while i0 > i+1:
            G1 = mvt(G1, 'B')
            LM.append('B')
            i0 -= 1
    return G1, LM

mvt_interdit = {"H": "B", "B": "H", "G": "D", "D": "G"}

#Q11.
def une_etape(Lk):
    Lk1 = []
    for couple in Lk:
        G, LM = couple
        if len(LM) == 0:
            interdit = None
        else:
            dernier_dep = LM[-1]
            interdit = mvt_interdit[dernier_dep]
        for dep in ['H','B','D','G']:
            i0, j0 = coord(G,0)
            if dep != interdit and EstLicite(i0, j0, n,
dep):
                G1 = mvt(G, dep)
                Lk1.append((G1,LM+[dep]))
    return Lk1
```