Exercice I (/14)

Partie 1 : Bit de parité

☐ **Q1 -** ◇ puisque $5 = \overline{101}^2$ le bit de parité est 0,

◇ puisque $16 = \overline{1\ 0000}^2$ le bit de parité est 1,

◇ puisque $37 = \overline{10\ 0101}^2$ le bit de parité est 1.

☐ **Q2 -**
```
def parite(bits : list) -> int :
    som = 0
    for b in bits :
        som += b
    return som % 2
```

☐ **Q3 -** On ne pourra pas détecter deux bits qui auront été intervertis ou qui auront changé de valeurs (*e.g.* $3 = \overline{000\ 0011}^2$ au lieu de $5 = \overline{000\ 0101}^2$ ou $0 = \overline{000\ 0000}^2$ au lieu de 5).
Si une erreur a été détectée, celle-i a pu advenir à 7 positions différentes ; il n'est pas possible de corriger sans retransmettre la donnée.

Partie 2 : Code de Hamming

☐ **Q4 -** On traduit l'encodage de Hamming exposé
```
def encode_hamming(donnee : list) -> list :
    d1, d2, d3, d4 = donnee
    p1 = parite( [d1, d2, d4] )
    p2 = parite( [d1, d3, d4] )
    p3 = parite( [d2, d3, d4] )
    return [p1, p2, d1, p3, d2, d3, d4]
```

☐ **Q5 -** On vérifie qu'il n'y ait pas d'erreur et on renvoie la donnée originelle, sinon on trouve l'indice de l'erreur, on la corrige puis on renvoie la donnée ainsi corrigée. Ce qui donne,
```
def decode_hamming(message : list) -> list :
    m1, m2, m3, m4, m5, m6, m7 = message
    c1 = parite( [m4, m5, m6, m7] )
    c2 = parite( [m2, m3, m6, m7] )
    c3 = parite( [m1, m3, m5, m7] )
    if c1 == 0 and c1 == c2 and c2 == c3 :
        return [m3, m5, m6 ,m7]
    bitNum = c1*4 + c2*2 + c3
    print( f"erreur à l'indice {bitNum}" )
    message[ bitNum-1 ] = 1 - message[ bitNum-1 ]
    return [ message[2] ] + message[4:]
```

☐ **Q6 -** Lorsque la donnée vaut `1011`, `(p1, p2, p3) = (0, 1, 0)`.
Ainsi le message encodé sera (0,1,1,0,0,1,1).

◇ Si les deux premiers bits ont été transmis incorrectement, le message reçu sera `1010011`. On aura alors `(c1, c2, c3) = (0, 0, 0)`.
Les deux erreurs simultanées n'auront pas été détectées et le message sera décodé par 1011 également (ce qui est attendu puisque seul les bits de parité ont été corrompus).

◇ Si les deux derniers bits (deux premiers de poids faibles ?) ont été transmis incorrectement, le message reçu sera `0110000`. On aura alors `(c1, c2, c3) = (0, 0, 0)`.
Les deux erreurs simultanées n'auront pas été détectées et le message sera décodé par 1000.

☐ **Q7 -** Comme suggeré, une possibilité pour palier la double erreur, il suffirait d'ajouter un 4è bit de parité au message entier (par exemple $m0$ en positon 0).

◇ S'il y a aucune erreur, le message décodé est toujours `(m3,m5,m6,m7)`.

◇ S'il y a une erreur, soit $m0$ est incorrect auxquel cas `(c1, c2, c3)` corrige l'erreur (puisqu'égal à (0,0,0)). soit $m0$ est correct, il existe une erreur dans les 7 bits du message que l'on sait corriger avec `(c1, c2, c3)`.

◇ S'il y a deux erreurs, soit $m0$ est incorrect, il existe une erreur dans les 7 bits du message que l'on sait corriger avec `(c1, c2, c3)`.

soit $m0$ est correct, on ne peut pas corriger les deux erreurs.

Exercice II (/12)

☐ **Q1 -**

```
def maxi(L) :
    Vmax = -1       # L contient des entiers naturels
    for i in range( len(L) ) :
        if Vmax < L[i] :
            Vmax = L[i]
    return Vmax
```

☐ **Q2 -**

```
def maxiR(L) :
    Vmax = -1       # L contient des entiers naturels
    if len(L) == 0 :
        return Vmax
    Vmax = maxiR( L[1:] )
    if L[0] > Vmax :
        return L[0]
    else :
        return Vmax
```

☐ **Q3 -**

```
def ind(L) :
    M = []
    for i in range( len(L) ) :
        if L[i] != 0 :
            M.append( i )
    return M
```

☐ **Q4 -**

```
def nb_oc(L) :
    M = maxi(L) + 1
    T = M * [0]
    for i in range( M ) :
        for j in range( len(L) ) :
            if L[j] == i :
                T[ i ] += 1
    return T
```

☐ **Q5 -** **a)** À chaque tour de boucle principale, on effectue `len(L)` tours de boucle secondaire. La boucle principale entraîne M tours.

Ainsi la liste `L` est parcourue $1 + M$ fois (1 fois dans `maxi` et M fois dans la boucle).

b)

```
def nb_oc(L) :
    M = maxi(L) + 1
    T = M * [0]
    for j in range( len(L) ) :
        T[ L[j] ] += 1
    return T
```

Dans ce cas, la liste `L` est parcourue deux fois (1 fois dans `maxi` et 1 fois dans la boucle).

☐ **Q6 -** **a)** $L_1 = [1,4,3,3,2,2,3,1,1,0]$; $L_2 = [1,4,3,3,2,2,3,1,1,0]$ et donc $L_1 = L_3 = L_{2018}$

b) La configuration $L_1 = [2,4,1,1,1,2]$ est impossible puisque les nombres 4, 1, 2 ne sont pas donnés dans l'ordre décroissant.

c) Si $L_1 = [2,4,1,0]$, L_0 est de longueur 3 et contient deux 4 et un 0.
Il peut y avoir 3 possibilités : $L_0 = [0,4,4]$, $L_0 = [4,0,4]$ et $L_0 = [4,4,0]$.

d)

```
def rob(A,n) :
    i = 0
    L = A[:]     # <=> A.copy()
    while i < n :
        T = nb_oc(L)
        I = ind(T)
        L = []
        for e in I :
            L.insert(0,e)   # insertion en début de liste
            L.insert(0,T[e])
            # ou lignes à remplacer par : L = [ T[e], e ] + L
        i += 1
    return L
```

Exercice III (/14)

g) On propose un code similaire à celui de chatGPT

```
def sum(tab : list) -> int :
    res = 0
    for i in range( len(tab) ) :
        res += tab[i]
    return res
```

h)

```
def operationF(mat : list, F : 'function' = lambda x : x) -> list :
    n, m = len(mat), len(mat[0])
    nmat = [ [0]*m for i in range(n) ]
    for i in range(n) :
        for j in range(m) :
            nmat[i][j] = F( mat[i][j] )
    return nmat
```

ou bien par compréhension

```
def operationF(mat : list, F : 'function' = lambda x : x) -> list :
    n, m = len(mat), len(mat[0])
    return = [ [ F( mat[i][j] ) for j in range(m)] for i in range(n) ]
```

i) Puisque la valeur `mat[i][j]` est l'indice du tableau qu'il faut incrémenter, on propose

```
def histogramme(mat : list) -> list :
    n, m  = len(mat), len(mat[0])
    hist = [0] * 256
    for i in range(n) :
        for j in range(m) :
            hist[ mat[i][j] ] += 1
    return hist
```

j) On propose

```
def indiceMinMax(hist : list) -> tuple :
    imin, imax = len(hist), -1
    for i in range( len(hist) ) :
        if hist[i] > 0 :
            if i < imin :
                imin = i
            if i > imax :
```

```
            imax = i
    return imin, imax
```

ou bien

```
def indiceMinMax(hist : list) -> tuple :
    n = len(hist)
    imin, imax = n, -1
    i = 0
    while i <= n//2 and ( imin == n or imax == -1 ) :
        if imin == n and hist[i] > 0 :
            imin = i
        if imax == -1 and hist[n-1-i] > 0 :
            imax = n-1-i
        i += 1
    return imin, imax
```

k) On définie f par $f(x) = \left\lfloor \frac{x-vmin}{vmax-vmin} * 255 \right\rfloor$.

```
def extension(x : int, hist : list) -> int :
    vmin, vmax = indiceMinMax(hist)
    return int( (x-vmin) / (vmax-vmin) * 255 )
```

l) L'appel de la fonction `operationF` qui permet de renvoyer la matrice codant pour l'image contrastée selon la méthode vue est `operationF(mat, extension)`

m)

```
def T(x : int, hist : list) -> int :
    N = sum(hist)
    return int( 255 / N * sum(hist[:x]) )
```

n) La fonction `extension` possède une complexité temporelle en $\mathcal{O}n$ tandis que la fonction `T` possède une complexité temporelle en $\mathcal{O}x + n$ (où $n = len(hist)$).

Si la fonction `indiceMinMax` est codée de manière optimale, hormis donner les-dites valeurs minimales et maximales de l'histogramme, on ne peut améliorer la fonction `extension`.

Concernant la fonction `T`, on peut améliorer le calcul de `sum(hist[:x])` au moment du calcul de `N=sum(hist)` en modifiant la fonction `sum` afin qu'il renvoie le tuple `sum(hist[:x])` et `sum(hist)`.

On peut proposer

```
def sum2(hist : list, x : int) -> tuple :
    # on suppose : 0 <= x <= len(hist)
    resX, res = 0, 0
    for i in range( x ) :
        resX += hist[i]
    for i in range( x, len(hist) ) :
        res += hist[i]
    return resX, res+resX
```

et ainsi obtenir pour la fonction `T` une complexité temporelle en $\mathcal{O}(n)$.

Rq : la complexité temporelle ne s'améliore pas significativement.