DS 4 - Concours Blanc Informatique

Simon Dauguet
simon.dauguet@gmail.com

Mercredi 06 Mars 2025

Exercice 1 :

Partie 1 : Bit de parité

1. En écrivant en base 2 :

- $5 = \overline{101}^2$, donc le bit de parité est 0,
- $16 = \overline{1\ 0000}^2$, donc le bit de parité est 1,
- $37 = \overline{10\ 0101}^2$, donc le bit de parité est 1.

2. On propose :

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

3. On ne pourra pas détecter deux bits qui auront été intervertis ou qui aurons changé de valeurs (par exemple $3 = \overline{000\ 0011}^2$ 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

4. 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]
```

5. 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,

```python
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:]
```

6. Lorsque la donnée vaut `1011`, on obtient `(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) = (`
Seul les bits de parité ont été corrompus mais l'algorithme conclu que le 3è bit est corrompu et le message sera décodé par 0011.

◇ 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.

7. Comme suggéré, 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 auquel 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 2 :

1. On propose :

```python
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
```

On pose n le nombre d'éléments de la liste `L`. Le booléen de la ligne 4 effectue 2 opérations. Et la ligne 5 également. Donc la structure `if` effectue $2 + \max(2, 0) = 4$ opérations dans le pire des cas. La fonction `maxi` effectue donc $1 + 4n + 1 = 4n + 2$ opérations.

La complexité de `maxi` est donc linéaire en $O(n)$.

2. On propose :

```
1  def ind(L) :
2      M = []
3      for i in range( len(L) ) :
4          if L[i] != 0 :
5              M += [i]     # <=> M.append( i )
6      return M
```

3. On propose :

```
1  def nb_oc(L) :
2      M = maxi(L) + 1
3      T = M * [0]
4      for i in range( M ) :
5          for j in range( len(L) ) :
6              if L[j] == i :
7                  T[ i ] += 1
8      return T
```

4. (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) On propose :

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

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

(c) La ligne 5 correspond à `T[L[j]] = T[L[j]]+1`, et donc effectue 6 opérations. En posant n le nombre d'éléments de la liste `L`, la boucle `for` effectue donc $6n$ opérations. La ligne 3 effectue $1 + (M - 1) = M$ opérations. Et la ligne 2 effectue $2 + O(n)$ opérations d'après le calcul de la complexité de la fonction `maxi`.

Par conséquent, la fonction `nb_oc` a une complexité linéaire en $O(n)$.

5. (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) On propose :

```
1      def rob(A,n) :
2          i = 0
3          L = A[:]     # <=> A.copy()
4          while i < n :
5              T = nb_oc(L)
6              I = ind(T)
7              L = []
8              for e in I :
9                  L.insert(0,e)    # insertion en début de liste
10                 L.insert(0,T[e])
11                 # ou lignes à remplacer par : L = [ T[e], e ] + L
12             i += 1
13         return L
```

(e) L'entier `n-i` est un variant de la boucle `while` de la fonction `rob`. En effet, à chaque passage dans la boucle `while`, l'algorithme passe par la ligne 12 et donc la variable `i` est incrémenté. Donc la valeur de `n-i` est une suite d'entier strictement décroissante. Et donc la boucle s'arrête bien.

Exercice 3 :

1. 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
```

2. On propose :

```
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) ]
```

3. 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
```

4. On propose :

```
def indiceMinMax(hist : list) -> tuple :
    i, n = 0, len(hist)
    while i < n and h[i] == 0:
        i += 1
    imin = i
    i = n-1
    while i >= 0 and h[i] == 0:
        i -= 1
    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
```

5. 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 )
```

6. 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)**

7. On propose :

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