BOUFALA Yacine

-20184427@etud.univ-evry.fr

KHALFOUN Rayan

-20183621@etud.univ-evry.fr

Entrée [ ]:
```
import os
import numpy as np
```

Entrée [ ]:
```
class __FileReader__:

    def __init__(self, path='Data/', output_path='Compressed/'):
        self.path = path
        self.output_path = output_path

    def read_txt(self, filename, splitter=False):
        """
        lire et renvoyer le contenu du fichier
        Decoupant le texte par rapport au espace (en mots)
        """
        data = open(self.path + filename, 'r', encoding='utf-8').read
        if splitter:
            return data.split(' ')
        return data


    def pad_encoded_text(self, encoded_text):
        extra_padding = 8 - len(encoded_text) % 8
        for i in range(extra_padding):
            encoded_text += "0"

        padded_info = "{0:08b}".format(extra_padding)
        encoded_text = padded_info + encoded_text
```

```
        return encoded_text


    def get_byte_array(self, padded_encoded_text):
        if(len(padded_encoded_text) % 8 != 0):
            print("Encoded text not padded properly")
            exit(0)

        b = bytearray()
        for i in range(0, len(padded_encoded_text), 8):
            byte = padded_encoded_text[i:i+8]
            b.append(int(byte, 2))
        return b


    def saveBin(self, filename, binCode):
        with open(self.output_path + filename + ".bin", 'wb') as outp
            padded_encoded_text = self.pad_encoded_text(binCode)
            b = self.get_byte_array(padded_encoded_text)
            output.write(bytes(b))
        return


    def read_dir(self, index=0):
        """
        recupere tout les fichier d'un dossier
        """
        return os.listdir(self.path.format(index))
```

Entrée [ ]:

```
class __RLE__:

    def __init__(self):
        self.m_RLE = []
        self.m_runs = 0
        self.m_size = 0

    def CreateRLE(self, p_array):
        currentrun = 0
        self.m_RLE.append(RLEPair(p_array[0], chr(1)))
        self.m_size = len(p_array) # the size of the uncompressed ar
        for index in range(1, self.m_size): # loops through each ite
            if p_array[index] != self.m_RLE[currentrun].m_data:
                currentrun+=1
                if self.m_size == currentrun:
                    self.m_size = currentrun * 2
                self.m_RLE.append(RLEPair(p_array[index], chr(1)))
            else:
                if self.m_RLE[currentrun].m_length == 255:
                    currentrun+=1
                    if self.m_size == currentrun:
                        self.m_size = currentrun * 2
                    self.m_RLE.append(RLEPair(p_array[index], 1))
                else:
                    self.m_RLE[currentrun].m_length = chr(ord(self.m_
        self.m_runs = currentrun + 1
```

```
    def FillArray(self, p_array):
        for currentrun in range(0, self.m_runs):
            for index in range(0, ord(self.m_RLE[currentrun].m_length
                p_array.append(self.m_RLE[currentrun].m_data)
```

Entrée [ ]:

```
class ArithmeticCoding:

    def __init__(self):
        self.__EOF__ = '<|*EOF*|>' # mot de fin de chaine
        self.precision = 16 # par defaut dans le papier


    def getfreqs(self, stream):
        """
        Cette fonction prend un flux en entrée. il obtient le nombre
        dans ce flux, stockez-le dans un dictionnaire contenant tous
        """
        symbols = set(stream) # supprimer les doublons dans le texte
        L = len(stream)

        Dict = {} # stocker chaque symbole avec sa probabilité corres

        # calculer la probabilité de chaque symbole dans le flux
        for s in symbols:
            freq = stream.count(s)
            Dict[s] = freq

        return Dict


    def Cumfreq(self, symbol, dictionary):
        """
        Cette fonction prend en entrée un symbole et un dictionnaire
        qui existent dans notre flux et leurs fréquenceset renvoie la
        tout début du Dictionnaire jusqu'à ce Symbole.
        """
        P = 0
        for sym in dictionary:
            P = P + dictionary[sym]
            if sym == symbol:
                break

        return P


    def Arithmetic_encode(self, stream):
        """
        La fonction d'encodeur pour le code arithmétique, toutes ses
        des nombres entiers
        """
        # ajouter le symbole de fin de fichier pour que le décodeur s
        stream.append(self.__EOF__)

        StreamSize = len(stream) # longeur de la liste

        dic = self.getfreqs(stream) # sortie le dictionnaire qui cont
```

```
        # borne superieur limite on la donne de base a (2^precision)
        # notre but c'est de la reduire au fur et a mesure jusqu'a av

        full = 2 ** self.precision
        half = full // 2 # la valeur milieu de la limite superieur
        quarter = half // 2 # la valeur du quart de la limite superie

        # notre interval ainsi contruit est donc [0, 2^precision[
        L = 0 # la limite inférieure de la plage
        H = full # la limite supérieure de la plage

        # Les zéros sont déplacés dans la partie basse de la plage de
        # les zéros sont décalés dans les bits de bas d'ordre inférie
        trails = 0 # Le shift

        code = [] # la liste qui contiendra le code compressé

        # De maniere iterative sur tout les mot de la liste qui const
        for symbol in stream:
            freqSym = dic[symbol] # obtenir la fréquence du symbole

            # on veux recuperer la frequence cumulé jusqu'a ce symbol
            # ce sera notre nouvelle borne superieur pour ce symbole

            # fonctionnement:
            # pour tout les symbol de notre dictionnaire qui existent
            # cumulée à partir du tout début du Dictionnaire jusqu'à
            # leurs frequence jusqu'a rencontrer le symbole en questi

            S_high = self.Cumfreq(symbol, dic) # obtenir la limite su

            S_low = S_high - freqSym # obtenir la limite inférieure d

            Range = H - L # obtenir la taille de plage de code

            H = L +  Range * S_high // StreamSize
            L = L +  Range * S_low // StreamSize

            # créer les cas pour lesquels on émettra 0 ou 1 à notre m
            while True: # les deux premiers cas faciles, si ma gamme

                if H < half : # si la plage complète tombe dans la mo
                    code.extend([0]) # on ecris un bit a 0 dans notre
                    code.extend([1] * trails) # on ecris autant de 1
                    trails = 0 # on reinitialise le trails a 0

                    # mettre à l'échelle la moitié inférieure pour êt
                    # on change les borne inf et sup car on est dans
                    # superieur H de la solution finale est superieur
                    # PS: on cherche a reduire L et H de sort a obter
                    # compression du texte
                    L = L * 2
                    H = H * 2

                elif L >= half: # si la plage complète tombe dans la
                    code.extend([1]) # on ecris un bit a 0 dans notre
                    code.extend([0] * trails) # on ecris autant de 0
                    trails = 0
```

```
                    # mettre à l'échelle la moitié supérieure pour êt
                    L = 2 * ( L - half )
                    H = 2 * ( H - half )

                # si la plage est divisée entre la moitié supérieure
                elif L >= quarter and H < 3 * quarter:
                    trails = trails + 1
                    L = 2 * ( L - quarter)
                    H = 2 * (H - quarter)
                else:
                    # elle n'est nulle part !
                    break

        # ajouter les derniers bits avant de quitter la fonction
        trails = trails + 1

        if L <= quarter: # la borne inf de la plage est dans le quart
            code.extend([0]) # on ecris un bit a 0 dans notre variabl
            code.extend([1] * trails) # on ecris autant de 1 que le c
        else:
            code.extend([1]) # on ecris un bit a 1 dans notre variabl
            code.extend([0] * trails) # on ecris autant de 0 que le c

        # une fois obtenue on retourne notre solution ecrite en binai
        # il est vrai que nous pouvons retourner l'interval [L, H] et
        # cette interval representera la compression du texte en entr
        return code, dic


    def Arithmetic_decode(self, code, dic):

        code_size = len(code)

        # obtenir la taille du flux acheter en additionnant le nombre
        stream_size = sum(dic.values())

        # ces nombres seront utilisés plus tard dans l'étape de mise
        full = 2 ** self.precision
        half = full // 2 # la valeur milieu de la limite superieur
        quarter = half // 2 # la valeur du quart de la limite superie

        # notre interval ainsi contruit est donc [0, 2^precision[
        L = 0 # la limite inférieure de la plage
        H = full # la limite supérieure de la plage

        # une variable qui contiendra la probabilité que nous itérero
        val = 0

        # un index pour tracer l'emplacement du dernier bit utilisé p
        # précision limitée dont nous disposons, car nous ne pouvons
        # entier en une seule fois
        indx = 1

        message=[]

        # obtenir d'abord le nombre exact de valeurs pouvant être cor
        # le reste sera utilisé pendant le code
        while indx <= self.precision and indx <= code_size:
            if code[indx - 1] == 1:
                val = val + 2 ** (self.precision - indx)
```

```
            indx = indx + 1

        # elle represente l'etat de la decompression (0=> trouvé , 1
        flag = 1

        # le drapeau sera nul si nous avons trouvé le symbole de fin
        while flag:

            # cette boucle essaie de trouver le symbole qui a la plag
            for symbol in dic :

                freqSym = dic[symbol] # obtenir la fréquence du symbo

                # on veux recuperer la frequence cumulé jusqu'a ce sy
                # ce sera notre nouvelle borne superieur pour ce symb

                # fonctionnement:
                # pour tout les symbol de notre dictionnaire qui exis
                # cumulée à partir du tout début du Dictionnaire jusq
                # fois leurs frequence jusqu'a rencontrer le symbole
                S_high = self.Cumfreq(symbol, dic) # obtenir la limit
                S_low = S_high - freqSym # obtenir la limite inférieu
                Range = H - L # obtenir la plage du code

                H0 = L + Range * S_high // stream_size
                L0 = L + Range * S_low // stream_size

                # si il est dans cette interval alors on a reussi a t
                if  L0 <= val and val < H0:
                    message.extend([symbol])
                    L = L0
                    H = H0

                    # cette condition doit être satisfaite à la fin d
                    if symbol == self.__EOF__ :
                        flag = 0 # on a trouvé le dernier mot alors d
                    break

            while True:
                if H < half : # si la plage est dans la moitié inféri
                    L = L * 2
                    H = H * 2
                    val = val * 2

                    if indx <= code_size:
                        val = val + code[indx - 1]
                        indx = indx + 1

                elif L >= half : # si la plage est dans la moitié sup
                    L = 2 * (L - half)
                    H = 2 * (H - half)
                    val = 2 * (val - half)

                    if indx <= code_size:
                        val = val + code[indx - 1]
                        indx = indx + 1

                # si la limite inférieure est dans la moitié inférieu
                #la moitié supérieure alors nous devrons élargir dava
                elif L >= quarter and H < 3 * quarter:
```

```
                    L = 2 * (L - quarter)
                    H = 2 * (H - quarter)
                    val = 2 * (val - quarter)

                    # augmenter la self.precision de la probabilité p
                    # tous les bits de code
                    if indx <= code_size :
                        val = val + code[indx - 1]
                        indx = indx + 1
                else:
                    break

        message.pop() # supprimer le symbole de fin de flux pour obte
        # retourne le message decompresser qui en toute logique est l
        return message
```

Entrée [ ]:

```
class UseCompressor:

    def __init__(self):
        self.arithmetic = ArithmeticCoding()
        self.file_ = __FileReader__()
        self.RLE = __RLE__()
        self. uncompressed = []
        self.texte = ''
        self.code = None
        self.dico = None


    def compress(self, path):
        try:
            filename, file_extension = os.path.splitext(path)
            self.texte = self.file_.read_txt(filename + file_extensio
            self.code , self.dico = self.arithmetic.Arithmetic_encode
            self.__RLE__.CreateRLE(''.join([str(x) for x in self.code
            self.file_.saveBin(filename, ''.join([str(x) for x in sel
            print('Compressed successfuly')
            return
        except:
            print('Compression Failed !')
            return


    def decompress(self):
        self.code = self.__RLE__().FillArray(self.code)
        message = self.arithmetic.Arithmetic_decode(self.code, self.c
        self.texte.pop()
        print('Decompression status: ', ' '.join(self.texte) == ' '.j
```

Entrée [ ]:

```
if __name__ == '__main__':
    algo = UseCompressor()
    filename = 'compression_3.txt'

    print('Compression...')
    algo.compress(filename)
    print('...')
    print('Decompression...')
```

```
    algo.decompress()
    print('...')
    x = os.stat('Data/' + filename)[6]
    print('Size (original): ', x, 'bytes')

    y = os.stat('Compressed/' + os.path.splitext(filename)[0]+ '.bin
    print('Size (compressed): ', y , 'bytes')

    print('Ratio: ', np.round(y*100 / x, 2))
```