Jose Luis Moreno Hernandez.
1000033265

Proyecto final de introducción a ciencias de la computación

Objetivo del proyecto:

Este proyecto se basa en la creación un sistema cuya finalidad es comprimir imágenes mediante el uso del algoritmo de compresión de Huffman.

Primero que todo vamos a aclarar un poco en que consiste el algoritmo de compresión de Huffman. [1]

- Contar cuantas veces aparece cada carácter en el fichero a comprimir. Y crear una lista enlazada con la información de caracteres y frecuencias.
- Ordenar la lista de menor a mayor en función de la frecuencia.
- Convertir cada elemento de la lista en un árbol.
- Fusionar todos estos árboles en uno único, para hacerlo se sigue el siguiente proceso, mientras la lista de árboles contenga más de un elemento:
    - Con los dos primeros árboles formar un nuevo árbol, cada uno de los árboles originales en una rama.
    - Sumar las frecuencias de cada rama en el nuevo elemento árbol.
    - Insertar el nuevo árbol en el lugar adecuado de la lista según la suma de frecuencias obtenida.
- Para asignar el nuevo código binario de cada carácter sólo hay que seguir el camino adecuado a través del árbol. Si se toma una rama cero, se añade un cero al código, si se toma una rama uno, se añade un uno.
- Se recodifica el fichero según los nuevos códigos.

Para la elaboración de este proyecto se programaron las siguientes clases y funciones:

- Clase nodo:

Jose Luis Moreno Hernandez.
1000033265

```
class nodo:
    def __init__(self,clave,valor,izquierdo=None,derecho=None,
                                          padre=None):
        self.clave = clave
        self.cargaUtil = valor
        self.hijoIzquierdo = izquierdo
        self.hijoDerecho = derecho
        self.padre = padre


    def esHijoIzquierdo(self):
        return self.padre and self.padre.hijoIzquierdo == self

    def esHijoDerecho(self):
        return self.padre and self.padre.hijoDerecho == self

    def esRaiz(self):
        return not self.padre
```

La finalidad de esta clase es ser soporte para los árboles binarios que se presentan a continuación.

- Clase Arbol:

Jose Luis Moreno Hernandez.
1000033265

```
class Arbol:
    def __init__(self,cant1,val1,cant2,val2):
        self.raiz = nodo(cant1+cant2,"x", None)
        self.raiz.hijoIzquierdo= nodo(cant2, val2,padre= self.raiz)
        self.raiz.hijoDerecho = nodo(cant1,val1,padre=self.raiz)


    def join(self,arbol2):
        nuevoNodo=nodo(self.raiz.clave+arbol2.raiz.clave,"x",None)
        self.raiz.padre=nuevoNodo
        nuevoNodo.hijoIzquierdo=arbol2.raiz
        nuevoNodo.hijoDerecho=self.raiz
        self.raiz=nuevoNodo

    def joinCant(self,cant1,val1):
        nuevoNodo=nodo(self.raiz.clave+cant1,"x",None)
        self.raiz.padre=nuevoNodo
        nuevoNodo.hijoIzquierdo=nodo(cant1,val1,padre=nuevoNodo)
        nuevoNodo.hijoDerecho= self.raiz
        self.raiz=nuevoNodo
```

En el código, esta clase sirve para implementar los árboles de Huffman. Los cuales son necesarios para obtener la forma en que se van a encriptar los caracteres. El constructor genera un árbol con tres nodos, el método join se encarga de unir dos árboles y el método joinCant se encarga de añadir un nuevo nodo al árbol. Los métodos nombrados anteriormente son necesarios para poder llevar a cabo el algoritmo de Huffman.

A continuación se presentan las funciones definidas como cortar y cortarn cuya finalidad es la misma, pero la diferencia radica en el resultado que se obtiene.

- Función cortarn:

Jose Luis Moreno Hernandez.
1000033265

```
69  def cortarn(matrizImagen):
70      copy = []
71      for i in range(0, len(matrizImagen), 2):
72          minicopy = []
73          for j in range(0, len(matrizImagen), 2):
74              minicopy.append( matrizImagen[i][j])
75          copy.append(minicopy)
76          minicopy = []
77      return copy
78
```

Esta función recibe como parámetro una matriz, la cual es una imagen y lo que hace es eliminar las filas y las columnas impares reduciendo su tamaño a un cuarto del tamaño original. La n en cortarn hace referencia a cortar normal.

- Función cortar:

```
57  def cortar(matrizImagen):
58      copy = []
59      for i in range(0, len(matrizImagen), 2):
60          minicopy = []
61          for j in range(0, len(matrizImagen), 2):
62              minicopy.append(" ".join(list(map(str,matrizImagen[i][j]))))
63
64          copy.append(" ".join(minicopy))
65
66      return copy
67
```

Esta función hace lo mismo que la anterior pero en lugar de retornar una matriz como tal, retorna una lista de las filas de la matriz como Strings

A continuación se presenta la contraparte de las funciones presentadas anteriormente:

- Función promedio:

```
96  def promediolineas(linea1,linea2):
97      lineaNueva=[]
98      for i in range(len(linea1)):
99          lineaNueva.append(promedio(linea1[i],linea2[i]))
100     return lineaNueva
101
```

Esta función se encarga de promediar dos triplas, donde cada tripla representa un color en RGB. Y se retorna la tripla resultante.

- Función promediolineas:

```
96  def promediolineas(linea1,linea2):
97      lineaNueva=[]
98      for i in range(len(linea1)):
99          lineaNueva.append(promedio(linea1[i],linea2[i]))
100     return lineaNueva
101
```

Esta función recibe dos filas de una matriz de imagen y crea una nueva fila compuesta por los promedios de las triplas de las filas que recibió, la cual retorna.

- Función descortar:

```
80  def descortar(matrizImagen):
81      matrizNueva=[]
82      for i in range(len(matrizImagen)):
83          miniNueva=[]
84          linea=matrizImagen[i]
85          for j in range(len(linea) - 1, 0, -1):
86              miniNueva.insert(0,linea[j])
87              prom = promedio(linea[j],linea[j - 1])
88              miniNueva.insert(0,prom)
89          matrizNueva.append(miniNueva)
90      matrizNuevaNueva=[]
91      for i in range(len(matrizNueva)-1,0,-1):
92          matrizNuevaNueva.insert(0,matrizNueva[i])
93          matrizNuevaNueva.insert(0,promediolineas(matrizNueva[i],matrizNueva[i-1]))
94      return matrizNuevaNueva
```

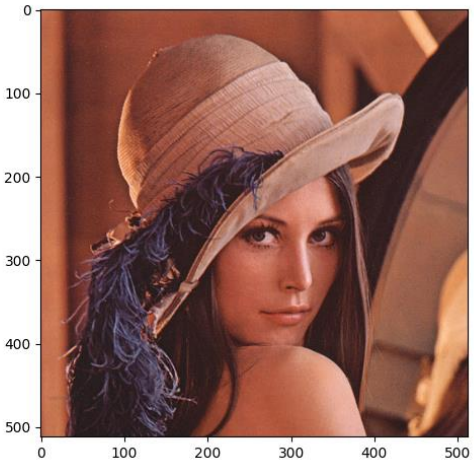
El propósito de esta función es revertir cortarn. Recibe como parámetro una matriz que haya pasado por la función cortarn e intenta restaurar las filas y columnas que fueron borradas insertando en esos espacios los promedios de las filas vecinas. Claramente no repone la imagen como era anteriormente pero si permite recuperar filas y columnas con cierto grado de precisión.

Las funciones presentadas anteriormente no tienen del todo que ver con el algoritmo de Huffman, tomémoslas como un plus para reducir tamaño. Ahora veamos un ejemplo de que es lo que hacen para entrar en contexto:

Jose Luis Moreno Hernandez.
1000033265

Como siempre vamos a trabajar con nuestra querida Lena, al ejecutar las siguientes líneas de código la obtenemos a ella.

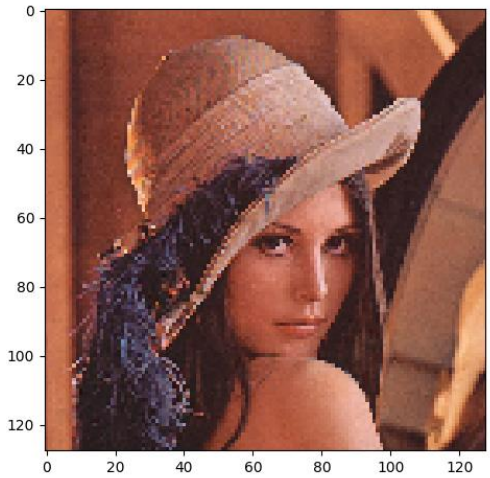
```
216  img = mpimg.imread('C:/Users/JOSE LUIS/Downloads/lena_color.BMP')
217  plt.imshow(img)
218  plt.show()
```



Ahora vamos a ver que ocurre cuando aplicamos dos veces la función cortarn:

```
216  img = mpimg.imread('C:/Users/JOSE LUIS/Downloads/lena_color.BMP')
217  img=cortarn(cortarn(img))
218  plt.imshow(img)
219  plt.show()
```

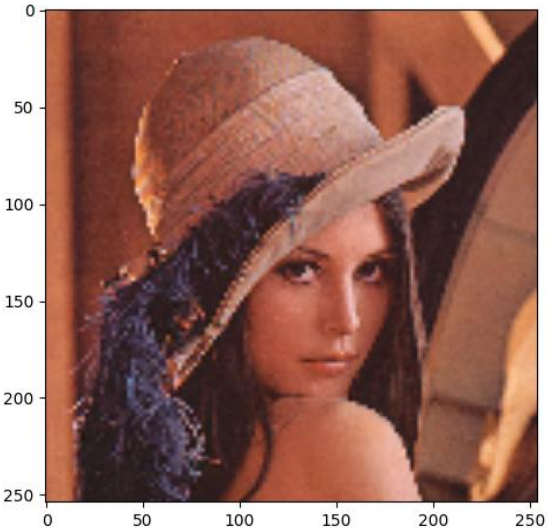
Como se puede ver en la línea 217 del código de arriba se aplicó dos veces la función, esto es lo que obtenemos:



Jose Luis Moreno Hernandez.
1000033265

Como se puede ver, se redujo considerablemente el tamaño y la calidad de Lena. Ahora vamos a aplicar la función descortar sobre la imagen obtenida a ver que ocurre:

```
216  img = mpimg.imread('C:/Users/JOSE LUIS/Downloads/lena_color.BMP')
217  img=cortarn(cortarn(img))
218  img=descortar(img)
219  plt.imshow(img)
220  plt.show()
```



Como podemos observar en el código de arriba se le aplica dos veces la función cortarn a la imagen y luego se le aplica una vez la función descortar, esto es lo que ocurre:

Como se pudo ver recupero tamaño y si comparamos mas a detalle se podrá observar que también recupera una calidad considerable:

Jose Luis Moreno Hernandez.
1000033265

La imagen de la izquierda es la cara de Lena tras solo haberle aplicado dos veces la función recortarn y la de la derecha es la imagen de la izquierda pero después de haberle aplicado la función desrecortar, se puede observar que se recupera una calidad considerable en la imagen.

Al hacer varias pruebas con las funciones mencionadas anteriormente se llegó a varias conclusiones sobre ese método. No es eficiente cuando las imágenes tienen bastante calidad ya que lo único que haría sería darle más tamaño casi con la misma calidad y tampoco es eficiente cuando la imagen tiene muy poca calidad ya que no habrían suficientes elementos en la matriz respectiva para que sea hecho de forma eficiente.

Ahora vamos a definir y explicar las funciones hechas para llevar a cabo la compresión con el algoritmo de Huffman.

- Función HuffmanArbol:

```
114  def HuffmanArbol(caracteres):
115      arbolP=Arbol(caracteres[0][1],caracteres[0][0],caracteres[1][1],caracteres[1][0])
116      cont=2
117
118      while cont<len(caracteres):
119          if caracteres[cont][1]>= arbolP.raiz.clave:
120              arbolP.joinCant(caracteres[cont][1],caracteres[cont][0])
121              cont+=1;
122          else:
123              if cont+2<=len(caracteres):
124                  newArbol=Arbol(caracteres[cont][1],caracteres[cont][0],caracteres[cont+1][1],caracteres[cont+1][0])
125                  arbolP.join(newArbol)
126                  cont+=2
127              else:
128                  arbolP.joinCant(caracteres[cont][1],caracteres[cont][0])
129                  cont+=1
130      return recorrerHuffman(arbolP.raiz,"")
```

  Esta función recibe un diccionario el cual contiene los caracteres presentes en la matriz de una imagen y la cantidad de veces que se repite cada uno. Mediante el uso de la clase Arbol mencionada anteriormente, genera un árbol siguiendo el algoritmo de Huffman donde todos los elementos del diccionario son hojas y aquellos con menor frecuencia tienen el camino mas largo. Retorna una llamada a la función recorrerHuffman que se describe a continuación.

- Función recorrerHuffman:

Jose Luis Moreno Hernandez.
1000033265

```python
dic = {}
dicre = {}
def recorrerHuffman(Nodo,bits):
    global dic
    global dicre
    if(Nodo.hijoDerecho==None and Nodo.hijoIzquierdo==None):
        dic[Nodo.cargaUtil]=bits
        dicre[bits]=Nodo.cargaUtil
    else:
        if(Nodo.hijoDerecho!=None):
            recorrerHuffman(Nodo.hijoDerecho,bits+"1")
        if (Nodo.hijoIzquierdo!= None):
            recorrerHuffman(Nodo.hijoIzquierdo, bits + "0")
    return dic,dicre
```

Esta función recibe la raíz de un árbol Huffman y lo que hace es crear las cadenas binarias correspondientes a cada carácter dependiendo del camino que deba recorrer para llegar a ellos. Retorna dos diccionarios, uno en el que asocia cada carácter con su respectiva cadena binaria y otro donde asocia cada cadena binaria con su respectivo carácter.

- Función binaryToLetter:

```python
def binaryToLetter(binaryString):
    mod=len(binaryString)%8
    finalString=""
    for i in range(0,len(binaryString)-mod,8):
        miniString=binaryString[i:i+8]
        asciicode=int(miniString,2)
        char=chr(asciicode)
        finalString+=char
    finalString+=binaryString[-mod:]
    return (finalString,mod)
```

Esta función recibe como parámetro una String binaria. Luego itera sobre ella recortando subStrings de longitud 8 las cuales transforma en su respectivo carácter de la tabla ascii y cada carácter de esos lo concatena formando una cadena la cual retorna.

- Función Huffman:

Jose Luis Moreno Hernandez.
1000033265

```
149  def Huffman(String):
150      cantidades={}
151      for i in String:
152          if i not in cantidades:
153              cantidades[i]=1
154          else:
155              cantidades[i]+=1
156      cantidades=sorted(cantidades.items(), key=operator.itemgetter(1))
157      BitsTable=HuffmanArbol(cantidades)
158      numBits=BitsTable[0]
159      Bitnum=BitsTable[1]
160      HuffmanString=""
161      for i in String:
162          HuffmanString+=numBits[i]
163      HuffmanString=binaryToLetter(HuffmanString)
164      return HuffmanString,numBits,Bitnum
165
```

Esta es la función principal donde se comprime la imagen. Recibe como parámetro la matriz de la respectiva imagen escrita como una cadena. Lo primero que hace es crear un diccionario donde guarda cada carácter de la cadena con su respectiva frecuencia y lo ordena de menor a mayor. Luego se llama la función HuffmanArbol con el respectivo diccionario y obtiene dos diccionarios los cuales son cada carácter con su representación binaria obtenida y viceversa. Seguido a ello crea una cadena vacia donde almacena la representación binaria de cada carácter de la cadena que representa la matriz y después llama a la función binaryToLetter obteniendo la cadena de caracteres que vendría siendo la imagen comprimida.

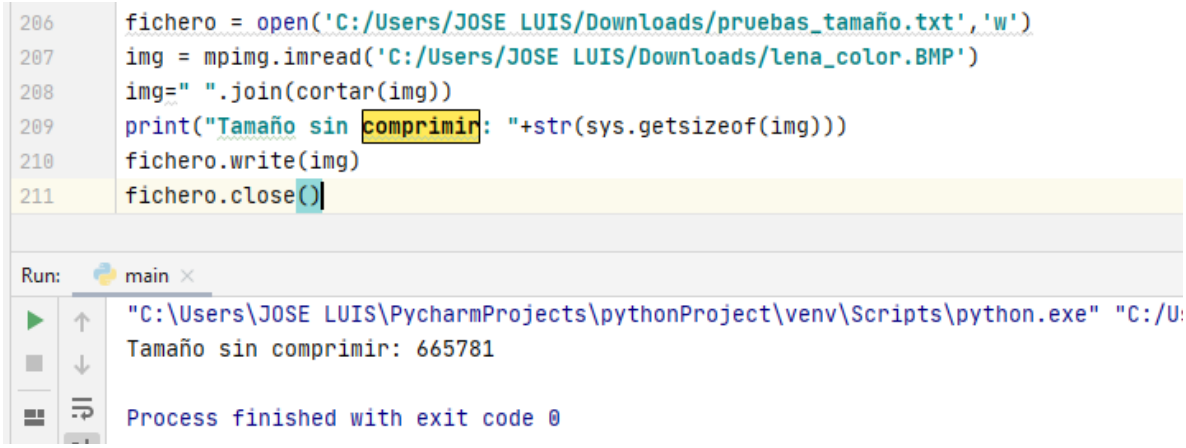
Ahora vamos a poner en práctica lo explicado anteriormente:

```
208  img = mpimg.imread('C:/Users/JOSE LUIS/Downloads/lena_color.BMP')
209  img=" ".join(cortar(img))
210  comprimida = Huffman(img)
211  print(comprimida[0])
```

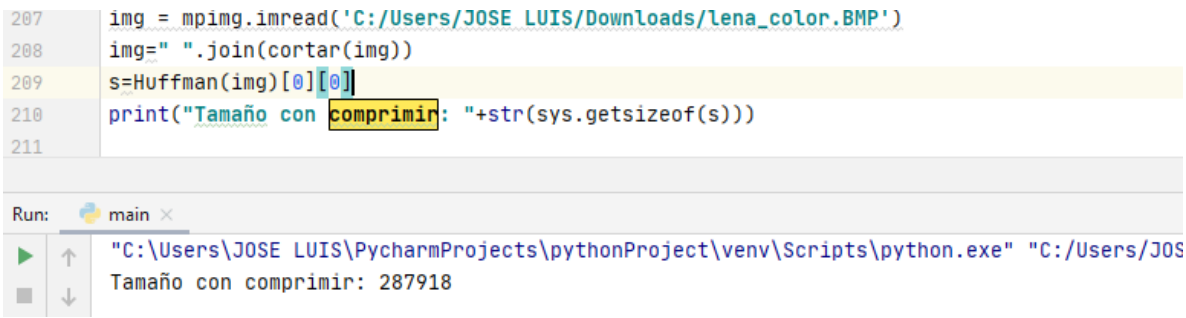
El código anterior convierte la imagen de Lena en una cadena y llama la función Huffman, así es como se ve la imagen comprimida al imprimirla.

Jose Luis Moreno Hernandez.
1000033265

Lo anterior es una parte de todo lo que se imprime. Ahora vamos a hablar en cuestión de memoria.



Si analizamos el tamaño de la cadena de la matriz habiendo con la función getsizeof de la librería sys después de haber aplicado la función cortar, obtenemos un resultado de 665781 bytes y al escribir la cadena en un archivo txt este tiene un tamaño de 651kb.



Jose Luis Moreno Hernandez.
1000033265

Al volver a analizar el tamaño de la cadena, pero esta vez pasando como parámetro una cadena comprimida por la función Huffman, obtenemos que el tamaño de esta es de 287918 bytes. Es decir, se redujo su tamaño aproximadamente en un 43%. Lastimosamente, no se logró almacenar dicha cadena un archivo txt ya que esta contenía caracteres que no son posible almacenarlos en ese tipo de archivos. Sin embargo, para hacernos una idea de cuanto ocuparía podemos hacer una regla de tres.

$$tamaño = \frac{651kb\ x\ 287918}{665781} \approx 281kb$$

Al comparar este resultado con el tamaño original de la imagen de Lena se obtiene una ganancia en espacio ya que el espacio ahorrado sería de casi 500kb. Debemos tener en cuenta que la imagen de Lena se encuentra en archivo BMP debido a que así se descargó. No se están teniendo en cuenta archivos JPG, PNG, etc.

Ya una vez comprimido, ahora debemos recrear la imagen a partir de la cadena de caracteres, para ello hacemos un proceso inverso de lo ya mencionado.

- Función toMatriz:

```
166  def toMatriz(numList):
167      numList=list(map(int,numList.split()))
168      matriz=[]
169      filas=[]
170      trio=[]
171      cont3=0
172      contFilas=0
173      for i in numList:
174
175          cont3+=1
176          trio.append(i)
177          if cont3==3:
178              filas.append(trio)
179              trio=[]
180              contFilas+=1
181              cont3=0
182              if contFilas==256:
183                  matriz.append(filas)
184                  filas=[]
185                  contFilas=0
186      return matriz
```

  Esta función recibe una cadena cuyo contenido son enteros separados por espacios y la convierte a una matriz en la cual cada posición contiene un color en RGB.

- Función desHuffman:

Jose Luis Moreno Hernandez.
1000033265

```
188 def desHuffman(HuffmanString,mod,bitDistribution):
189     binarynoHuffmanString=""
190     for i in range(len(HuffmanString)-mod):
191         binarynoHuffmanString+=inttoBin(ord(HuffmanString[i]))
192     binarynoHuffmanString+=HuffmanString[-mod:]
193     noHuffmanString=""
194     controler=""
195     for i in binarynoHuffmanString:
196         controler+=i
197         if controler in bitDistribution:
198             noHuffmanString+=bitDistribution[controler]
199             controler=""
200     matriz=toMatriz(noHuffmanString)
201     return matriz
202
```

Esta función viene siendo la inversa a la función Huffman. Recibe como parámetros la cadena que deseamos descomprimir y un diccionario donde se encuentran las cadenas binarias con sus respectivos caracteres. Primero convertimos cada carácter a su representación binaria en ascii y lo concatenamos a otra cadena, luego iteramos sobre dicha cadena comparando cuales subcadenas se encuentran en el diccionario para convertir esas subcadenas binarias a los caracteres que van a estar presentes en la matriz de la imagen. Al obtener la cadena de caracteres que van a estar en la matriz llamamos a la función toMatriz la cual va a convertir esa cadena en una matriz y por último se retorna dicha matriz.

Ahora vamos a comprimir y descomprimir una imagen para ver que funciona:

```
207 img = mpimg.imread('C:/Users/JOSE LUIS/Downloads/lena_color.BMP')
208 img=" ".join(cortar(img))
209 s=Huffman(img)
210 nuevaMatriz=desHuffman(s[0][0],s[0][1],s[2])
211 print(nuevaMatriz)
```

```
Run: main
"C:\Users\JOSE LUIS\PycharmProjects\pythonProject\venv\Scripts\python.exe" "C:/Users/JOSE LUIS/PycharmProjects/pythonProject/main.py"
[[[193, 111, 74], [195, 113, 76], [200, 116, 80], [199, 113, 78], [197, 112, 75], [196, 111, 74], [192, 106, 71], [190, 104, 71], [186, 101, 64], [182,
 96, 61], [176, 91, 60], [165, 81, 53], [139, 64, 43], [127, 58, 42], [113, 55, 44], [103, 54, 49], [93, 45, 41], [86, 39, 33], [102, 53, 46], [97, 48,
 41], [112, 53, 47], [109, 50, 42], [112, 54, 43], [114, 56, 45], [115, 56, 48], [116, 56, 46], [117, 57, 46], [124, 62, 51], [114, 54, 44], [117, 57,
 47], [120, 60, 50], [114, 54, 44], [118, 66, 52], [132, 65, 56], [118, 64, 52], [118, 61, 50], [123, 71, 60], [124, 65, 49], [128, 63, 43], [132, 71,
 53], [129, 59, 49], [134, 65, 49], [139, 71, 50], [146, 77, 62], [144, 74, 62], [150, 78, 64], [151, 78, 61], [154, 81, 62], [152, 80, 58], [150, 78,
 56], [156, 81, 60], [149, 72, 52], [158, 83, 60], [161, 86, 63], [158, 83, 60], [153, 78, 55], [152, 80, 55], [158, 83, 60], [161, 86, 65], [155, 80,
 59], [161, 85, 62], [159, 83, 60], [160, 84, 61], [160, 84, 61], [161, 85, 61], [157, 81, 57], [162, 84, 61], [158, 78, 55], [162, 85, 59], [158, 81,
 55], [164, 87, 61], [163, 86, 60], [167, 87, 64], [168, 88, 65], [166, 84, 62], [167, 82, 61], [165, 83, 61], [164, 82, 60], [164, 82, 60], [167, 85,
 63], [163, 83, 60], [162, 82, 59], [166, 86, 63], [161, 81, 58], [159, 82, 56], [163, 85, 62], [163, 85, 63], [161, 83, 61], [156, 75, 54], [159, 78,
 57], [161, 80, 59], [165, 84, 63], [155, 75, 52], [162, 82, 59], [153, 73, 50], [162, 82, 59], [161, 85, 62], [161, 83, 61], [161, 80, 59], [167, 85,
 64], [159, 82, 56], [163, 85, 62], [164, 86, 63], [157, 79, 57], [165, 85, 60], [154, 76, 53], [161, 83, 60], [158, 82, 59], [165, 85, 58], [163, 86,
 60], [160, 83, 57], [157, 81, 57], [158, 86, 62], [161, 86, 63], [164, 88, 65], [163, 85, 63], [160, 86, 61], [156, 80, 56], [168, 91, 65], [160, 83,
 55], [162, 86, 62], [157, 81, 57], [160, 84, 60], [156, 80, 56], [158, 86, 61], [159, 85, 60], [157, 81, 57], [157, 81, 57], [161, 80, 59], [159, 81,
 59], [154, 78, 55], [162, 87, 64], [155, 77, 55], [155, 77, 55], [163, 87, 63], [162, 86, 60], [155, 81, 56], [156, 82, 57], [157, 81, 57], [160, 82,
 59], [149, 78, 56], [154, 82, 58], [161, 87, 62], [155, 79, 53], [159, 87, 63], [154, 82, 58], [156, 84, 60], [163, 91, 67], [155, 81, 56], [157, 83,
 58], [154, 82, 58], [156, 85, 63], [152, 88, 63], [146, 82, 57], [145, 77, 54], [152, 80, 58], [135, 76, 58], [136, 74, 51], [140, 70, 45], [140, 71,
 55], [132, 67, 49], [132, 69, 52], [131, 69, 54], [119, 61, 49], [114, 56, 44], [120, 61, 47], [129, 68, 50], [142, 80, 59], [155, 87, 64], [175, 105,
```

Como podemos ver, se ha comprimido con la función Huffman y se ha descomprimido con la función desHuffman y se puede ver que hemos recuperado la matriz inicial. Ahora vamos a observar el resultado final al mostrar la imagen.

Jose Luis Moreno Hernandez.
1000033265

```
#fichero = open('C:/Users/JOSE LUIS/Downloads/pruebas_tamaño.txt','w')
img = mpimg.imread('C:/Users/JOSE LUIS/Downloads/lena_color.BMP')
img=" ".join(cortar(img))
s=Huffman(img)
nuevaMatriz=desHuffman(s[0][0],s[0][1],s[2])
nuevaMatriz=descortar(nuevaMatriz)
plt.imshow(nuevaMatriz)
plt.show()
```

Como resultado final obtenemos a Lena casi con la misma calidad que como la leemos. Por lo que se puede concluir que el algoritmo aplicado en el presente proyecto genera rentabilidad a la hora de conservar espacio en memoria y no solo es aplicable a imágenes sino a cualquier objeto que de alguna manera se pueda escribir como una cadena.

Referencias:

[1] Tomado de: http://conclase.net/blog/item/huffman