# DIP-A2-G10

- **Q1 - HUFFMAN ENCODING AND DECODING**
- **Q2 - ARITHMETIC CODING**

# SUMMARY

**QUESTION-1: Write a generic function to perform Huffman Coding and Decoding on Image or Text.**

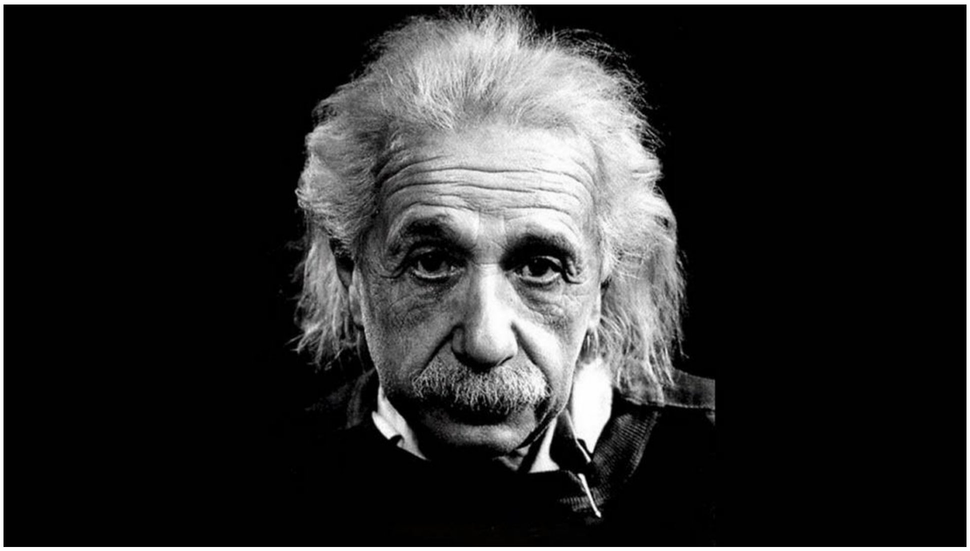
Language/Libraries Used: → Python
→ NumPy
→ Pillow (PIL)
→ Regular Expression (re)

Description of Input Variables:

- **choice:** choosing between input as an image (1) or text (2) otherwise "invalid input".
- **my_string:** if choice==1, we will take input, an Image and if choice==2, we will take input, a dictionary/matrix with 2 columns (symbol and it's probability).
- **message_to_encode**: optional parameter to the function. If provided with value then encoded message to be printed using generated Huffman code for the text and if not provided, then function should return the result of Huffman coding on single channel image.

Example Inputs:

1. **Image:** (1920 × 1080 dimension)



2. **Text:**

Dictionary= {'a': 0.25, 'b': 0.25, 'c': 0.2, 'd': 0.15, 'e': 0.15}

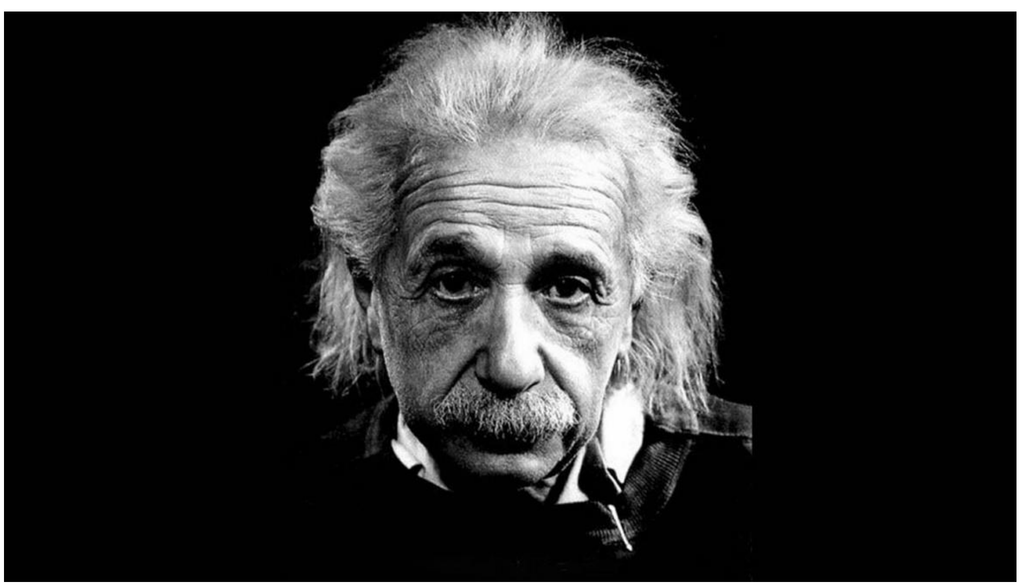
## Expected Output:

- **Probability table:** probability table used for encoding to be printed.
- **Code book generated:** print code-word for each symbol generated by Huffman encoding.
- **compressed.txt:** store result of applying Huffman Coding on the specified data, specified as a string of ‘0’ and ‘1’ in .txt file.
- **Decoding:**

  a. when *message_to_encode* parameter is passed, encoded message to be printed for the given text:

     encoded_message for ['a', 'b', 'a', 'c'] ======> 01100100

  b. else decoded image to be generated for given single channel image:



# OUTPUT ANALYSIS

When Image is given as an Input:

```
C:\Users\vijay\miniconda3\python.exe C:/Users/vijay/Desktop/Huffman-Encoding-Decoding/huffman_final.py

Enter 1 for a single channel image compression or 2 for text compression: 1

Probability table used: [(' ', 0.29), (',', 0.29), ('0', 0.02), ('1', 0.25), ('2', 0.04), ('3', 0.02), ('4', 0.02), ('5', 0.02), ('6', 0.01), ('7', 0.01), ('8', 0.01), ('9', 0.01), ('[', 0.0), (']', 0.0)]

code book generated:
Symbol code-word
 [      00111100
 1      01
 ,      11
        10
 0      00000
 2      0001
 4      00100
 3      00001
 8      001101
 5      00101
 9      001110
 6      0011111
 7      001100
 ]      00111101

Compressed file generated as compressed.txt

Started Decoding.......
Input image dimensions: (1080, 1920)
Output image dimensions: (1080, 1920)
Decoded successfully

Process finished with exit code 0
```

When dictionary/matrix of 2 columns (symbols and their probabilities) is given:

```
C:\Users\vijay\miniconda3\python.exe C:/Users/vijay/Desktop/Huffman-
Encoding-Decoding/huffman_final.py

Enter 1 for a single channel image compression or 2 for text
compression: 2

Probability table used: {'a': 0.25, 'b': 0.25, 'c': 0.2, 'd': 0.15,
'e': 0.15}

merging lowest probability pair ['d', 'e']  ======>
[(0.2, ['c']), (0.25, ['a']), (0.25, ['b']), (0.3, ['d', 'e'])]
merging lowest probability pair ['c', 'a']  ======>
[(0.25, ['b']), (0.3, ['d', 'e']), (0.45, ['c', 'a'])]
merging lowest probability pair ['b', 'd', 'e']  ======>
[(0.45, ['c', 'a']), (0.55, ['b', 'd', 'e'])]
merging lowest probability pair ['c', 'a', 'b', 'd', 'e']  ======>
[(1.0, ['c', 'a', 'b', 'd', 'e'])]

encoded_message for ['a', 'b', 'a', 'c']  ======>  01100100

code book generated:
Symbol code-word
 a      01
 b      10
 c      00
 d      110
 e      111

Process finished with exit code 0
```

# RESULTS

- Given image as an input, can be compressed using Huffman lossless compression algorithm to save space and to perform operations on the images faster. Huffman coding uses frequency of the symbols to generate a tree and perform compression based on it. Later on, the same compressed form of an image can be decoded using the code book generated by Huffman encoding.

  Image compression can be done using many ways. Instead of taking histogram and normalizing it, we have taken image matrix and converted it into string. By converting, we can see that we only need 14 levels to generate Huffman tree which is optimized way in terms of both space as well as time. Below are the only unique symbols required for the Huffman tree encoding instead of 256 unique pixel values:

  **['[', '1', ',', ' ', '0', '2', '4', '3', '8', '5', '9', '6', '7', ']']**

- Same operations can be performed on text as well. Just that frequency of letters in the text is used to generate Huffman tree.

## QUESTION-2: Write a generic function to perform Arithmetic Coding on Image or Text.

Language/Libraries Used: → Python
→ NumPy
→ CV2 (OpenCV)

Description of Input Variables:

- **sym_prob_mat**: Either “A matrix with two columns (symbol number and its probability)” or “A single channel image”.
- **N**: A number ‘N’ which indicates the number of symbols.
- **message**: The message to be coded as a 1-D array.

## Example Inputs:

1. **Text**: Message = “sarangvijaynitin”

```
In [11]: %run arithmetic_coding_Q2.py

         Enter 1 for a single channel image encoding or 2 for text encoding: 2
         The Symbol Probability Input Matrix :

         [['s' '0.0625']
          ['a' '0.1875']
          ['r' '0.0625']
          ['n' '0.1875']
          ['g' '0.0625']
          ['v' '0.0625']
          ['i' '0.1875']
          ['j' '0.0625']
          ['y' '0.0625']
          ['t' '0.0625']]

         Number of unique symbols are :

         10

         Message to Encode :


         ['s', 'a', 'r', 'a', 'n', 'g', 'v', 'i', 'j', 'a', 'y', 'n', 'i', 't', 'i', 'n']
```

2. **Image**: Single channel image of 50 × 50 pixels

## Expected Output:

- A structured table showing the symbols along with their probabilities for each stage of iteration.
- Sub-message and its corresponding probability ranges(code) for each stage of iteration.

# OUTPUT ANALYSIS

When Image is given as an Input:

For the single channel input image of 50 × 50 pixels, we get an output of 2500 iterations.

Starting from actual output image in 1st iteration,

```
In [6]: %run arithmetic_coding.py

        Enter 1 for a single channel image encoding or 2 for text encoding: 1
        Stage_1
             symbol  probability  length    from      to
        0       0.0       0.0648  0.0648  0.0000  0.0648
        1       1.0       0.0072  0.0072  0.0648  0.0720
        2       2.0       0.0068  0.0068  0.0720  0.0788
        3       3.0       0.0076  0.0076  0.0788  0.0864
        4       4.0       0.0088  0.0088  0.0864  0.0952
        ..      ...          ...     ...     ...     ...
        251   251.0       0.0004  0.0004  0.9924  0.9928
        252   252.0       0.0004  0.0004  0.9928  0.9932
        253   253.0       0.0008  0.0008  0.9932  0.9940
        254   254.0       0.0008  0.0008  0.9940  0.9948
        255   255.0       0.0052  0.0052  0.9948  1.0000

        [256 rows x 5 columns]

        Message : [[3]]  ::  Code : [0.0788 to 0.08639999999999999)
```

Up to the 2500th iteration,

```
Stage_2500
     symbol  probability  length    from      to
0       0.0       0.0648     0.0  0.0788  0.0788
1       1.0       0.0072     0.0  0.0788  0.0788
2       2.0       0.0068     0.0  0.0788  0.0788
3       3.0       0.0076     0.0  0.0788  0.0788
4       4.0       0.0088     0.0  0.0788  0.0788
..      ...          ...     ...     ...     ...
251   251.0       0.0004     0.0  0.0788  0.0788
252   252.0       0.0004     0.0  0.0788  0.0788
253   253.0       0.0008     0.0  0.0788  0.0788
254   254.0       0.0008     0.0  0.0788  0.0788
255   255.0       0.0052     0.0  0.0788  0.0788

[256 rows x 5 columns]

Message : [[ 3]
 [ 0]
 [ 0]
 ...
 [19]
 [53]
 [21]]  ::  Code : [0.07880001063969881 to 0.07880001063969881)
```

When a matrix of 2 columns (symbols and their probabilities) is given as input:

For the input message [‘s’,’a’,’r’,’a’,’n’,’g’,’v’,’i’,’j’,’a’,’y’,’n’,’i’,’t’,’i’,’n’]

The actual output starts from 1st iteration,

```
Message to Encode :


['s', 'a', 'r', 'a', 'n', 'g', 'v', 'i', 'j', 'a', 'y', 'n', 'i', 't', 'i', 'n']

Stage_1
  symbol probability  length    from      to
0      s      0.0625  0.0625  0.0000  0.0625
1      a      0.1875  0.1875  0.0625  0.2500
2      r      0.0625  0.0625  0.2500  0.3125
3      n      0.1875  0.1875  0.3125  0.5000
4      g      0.0625  0.0625  0.5000  0.5625
5      v      0.0625  0.0625  0.5625  0.6250
6      i      0.1875  0.1875  0.6250  0.8125
7      j      0.0625  0.0625  0.8125  0.8750
8      y      0.0625  0.0625  0.8750  0.9375
9      t      0.0625  0.0625  0.9375  1.0000

Message : ['s']  ::  Code : [0 to 0.0625)
```

Up to the 16th iteration,

```
Stage_16
  symbol probability        length      from        to
0      s      0.0625  3.556725e-16  0.006938  0.006938
1      a      0.1875  1.067018e-15  0.006938  0.006938
2      r      0.0625  3.556725e-16  0.006938  0.006938
3      n      0.1875  1.067018e-15  0.006938  0.006938
4      g      0.0625  3.556725e-16  0.006938  0.006938
5      v      0.0625  3.556725e-16  0.006938  0.006938
6      i      0.1875  1.067018e-15  0.006938  0.006938
7      j      0.0625  3.556725e-16  0.006938  0.006938
8      y      0.0625  3.556725e-16  0.006938  0.006938
9      t      0.0625  3.556725e-16  0.006938  0.006938

Message : ['s', 'a', 'r', 'a', 'n', 'g', 'v', 'i', 'j', 'a', 'y', 'n', 'i', 't', 'i', 'n']  ::  Code : [0.006938487521141956 to
0.006938487521143023)
```

## RESULTS

- In Arithmetic coding, the message is encoded as a real number between zero to one.

- It gives a range of probability as output, any value within that probability range corresponds to the original message.

- Unlike Huffman, in arithmetic coding entire message, that is to be encoded, must be present in order to start encoding.

- There is a limit to the precision of number that is to be encoded. Therefore, there is a limit to the length of message that is to be encoded.