TASK 3: The file Challange.txt would not be efficiently encoded using HUFFMAN ENCODING for that case you can use any efficient data compression technique you have to code it by yourself and you should be able to explain why it works fine on this file

While running tests on Huffman Encoding on "Challange.txt" in the previous part, we noticed that the algorithm was not that much efficient as Challange.txt had numerous repeated characters continuously, which made Huffman a bit inefficient. So, we had to come up with a more efficient technique. We researched for 3-4 days and nights and shortlisted multiple algorithms such as RLE compression, LZ78, LZ77, Arithmetic coding, Sequitur etc. After detailed analysis and running every algorithm, we finally decided to move forward with LZW Data Compression Algorithm.

LZW Encoding Algorithm

LZW stands for **"Lempel-Ziv-Welch"**. The LZW algorithm is a very common compression technique. This algorithm is typically used in GIF and PDF. It is lossless, meaning no data is lost when compressing. The algorithm is simple to implement and has the potential for very high throughput in hardware implementations. It is the algorithm of the widely used Unix file compression utility compress, and is used in the GIF image format.

This algorithm basically utilizes reoccurring patterns to save data space. As our "Challange.txt" file had numerous repeating characters. So, this algorithm will be most efficient for that file. LZW compression works by reading a sequence of symbols, grouping the symbols into strings, and converting the strings into codes.

LZW compression uses a code table, with 4096 as a common choice for the number of table entries. Codes 0-255 in the code table are always assigned to represent single bytes from the input file. When encoding begins the code table contains only the first 256 entries, with the remainder of the table being blanks. Compression is achieved by using codes 256 through 4095 to represent sequences of bytes. As the encoding continues, LZW identifies repeated sequences in the data, and adds them to the code table. Decoding is achieved by taking each code from the compressed file and translating it through the code table to find what character or characters it represents. Following picture is the explanation of the encoding process that we borrowed from https://www2.cs.duke.edu/csed/curious/compression/lzw.html to understand the algorithm.

Input	Current String	Seen this Before?	Encoded Output	New Dictionary Entry/Index
b	b	yes	nothing	none
ba	ba	no	1	ba / 5
ban	an	no	1,0	an / 6
bana	na	no	1,0,3	na / 7
banan	an	yes	no change	none
banana	ana	no	1,0,3,6	ana / 8
banana_	a_	no	1,0,3,6,0	a_ / 9
banana_b	_b	no	1,0,3,6,0,4	_b / 10
banana_ba	ba	yes	no change	none
banana_ban	ban	no	1,0,3,6,0,4,5	ban / 11
banana_band	nd	no	1,0,3,6,0,4,5,3	nd / 12
banana_banda	da	no	1,0,3,6,0,4,5,3,2	da / 13
banana_bandan	an	yes	no change	none
banana_bandana	ana	yes	1,0,3,6,0,4,5,3,2,8	none

For the implementation of this algorithm, we used all the concepts of Object-Oriented Programming to divide a problem into different parts and then utilize each part where ever necessary. For this implementation, we have 6 different files and 1 main file to call each of them. The breakdown of the files is as follows:

1. bitstream.h
2. bitstream.cpp
3. dictionary.h
4. dictionary.cpp
5. lzw.h
6. lzw.cpp
7. main.cpp

We then performed tests using two of test files named ***"file1.txt"*** and ***"Challange.txt"***. Following is the results after running the tests. *(For most accurate results, we ran each file 3 times and calculated its average and store it)*

To run the file, follow the following steps:

- g++ main.cpp
- ./a.out

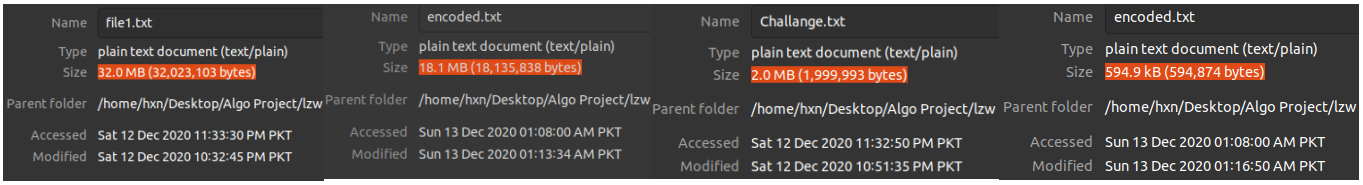
It will then ask you to enter the name of the file. Enter the name **without the extension (i.e., ".txt")**. Enter the name of the file you want to encrypt.

Results:

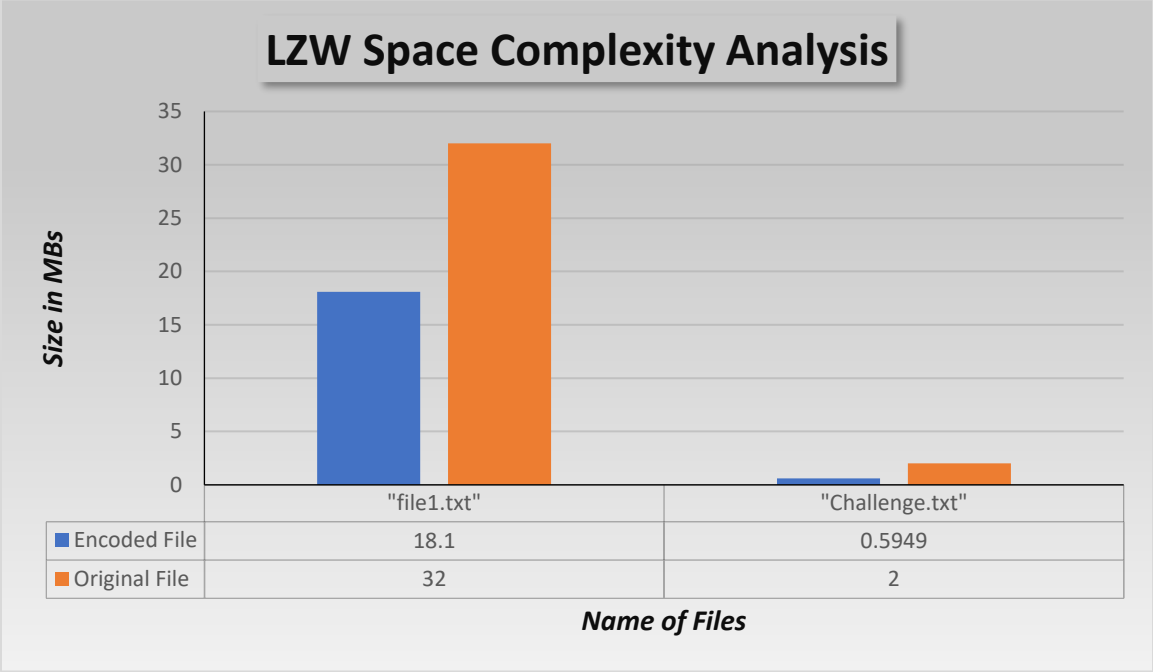
The results of the tests are as follows:

Name of the file	Original Size (MBs)	Size after Encryption (MBs)	% Compressed	Time for Encryption
"file1.txt"	32.0 (32,023,103B)	18.1 (18,135,838B)	43.36%	164.3s (2.74 min)
"Challange.txt"	2.0 (1,999,993B)	0.594 (594,874B)	70.26%	8.3s (0.14 mins)

Note: The "Time of Encryption" includes the time taken to read the test file.

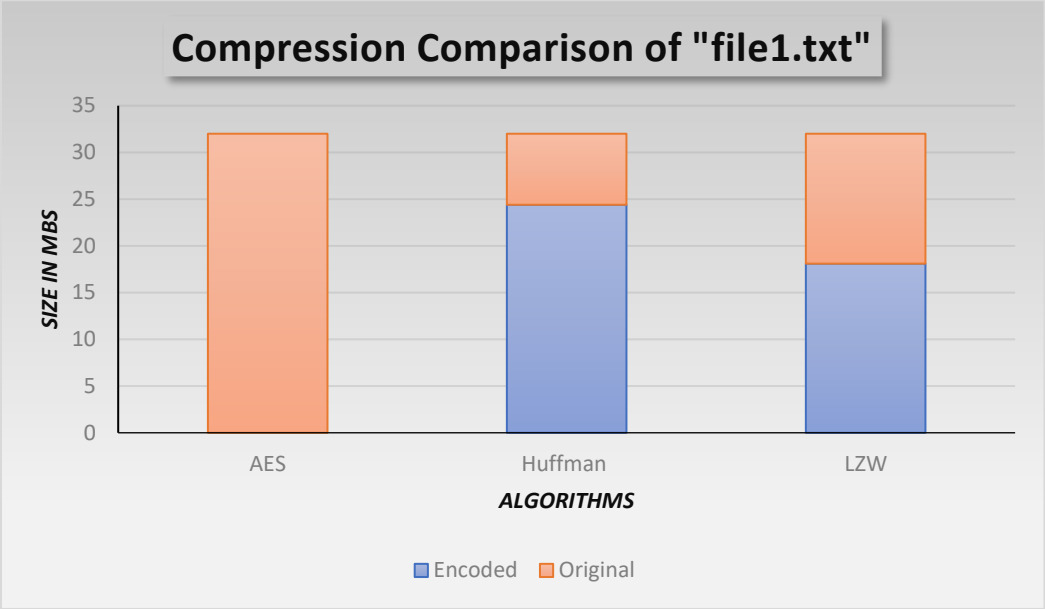


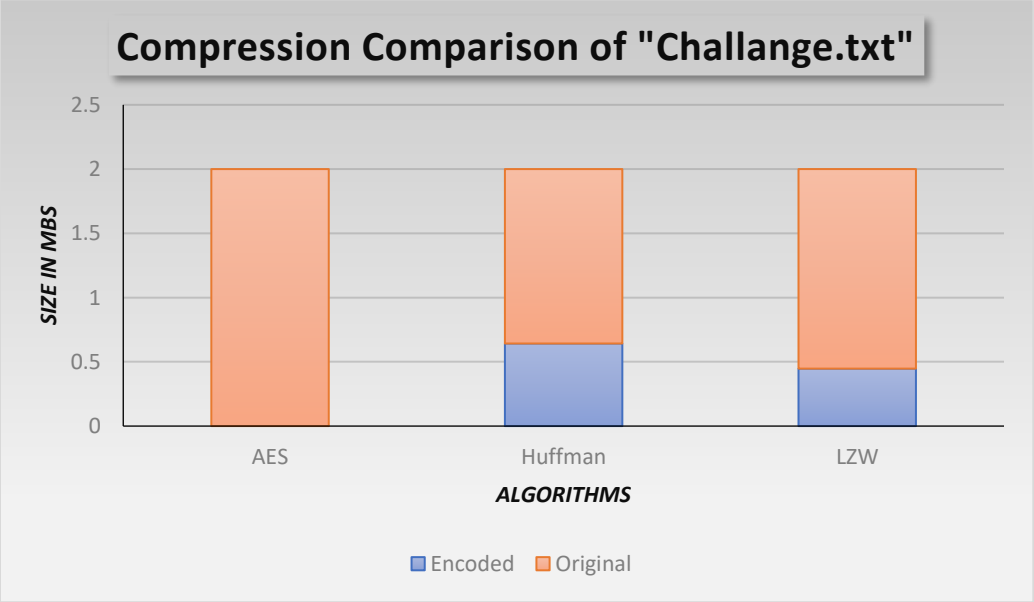
LZW Encoding Algorithm Benchmarks:



As can be seen from the above graph, LZW Compression Algorithm showed remarkable performance in terms of space complexity and time both. LZW performed better on "Challange.txt" as the file contained multiple reoccurring characters which are handled more efficiently by LZW as compared to Huffman.

Comparison in terms of Space:

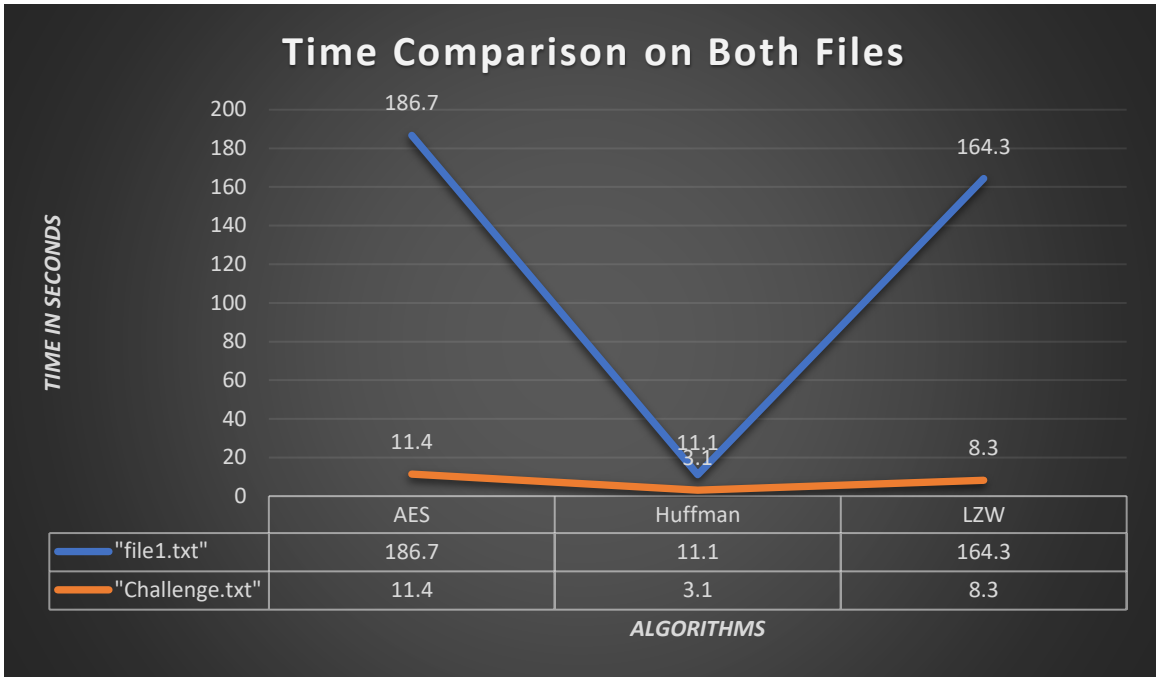




From the above-mentioned graphs, it can be easily deduced that AES algorithm has the worst performance in terms of space. Comparing the remaining two, we can see LZW winning with a major difference. This means that in terms of space, LZW Data Compression is the most efficient of all three.

Note: In case of Huffman and LZW, the smaller the blue bar, the higher the space efficiency.

Comparison in terms of Time:



From the above-mentioned graph, it can be easily deduced that AES algorithm has the worst performance in terms of space. Comparing the remaining two, we can see Huffman winning with a major difference. This means that in terms of time, Huffman Data Encryption Algorithm is the most efficient of all three.

After analyzing all of the above result, we can see that Huffman data compression is time efficient whereas LZW data compression is space efficient. But as the project was about data compression, so LZW Data Compression Algorithm will be considered as the overall best algorithm for our data compression.

THE END 😊
