جامعة الأمير سلطان
PRINCE SULTAN UNIVERSITY

DESIGN & ANALYSIS OF ALGORITHMS

LOSSLESS COMPRESSION ALGORITHMS

CS311 PROJECT

TEAM MEMBERS:

Reem Hejazi 219410002

Raneem Balharith 219410305

Sara Al Shagawi 219410319

SUPERVISED BY:

Dr. Huda Alrammah

TABLE OF CONTENTS

INTRODUCTION:

With the increase in the use of the Internet, data communication now is an essential topic in Computer Science, and here comes the role of compression algorithms. **Data compression** is the process of encoding, restructuring data in order to reduce its size. To reduce both the time and space expenses in the data communication process.

Mainly, there are two types of data compression algorithms:
-Lossless compression: removes the statistical redundancies while maintaining the data, so no data loss is present. **PNG**, **FLAC**, and **ZIP** use lossless compression algorithms.
-Lossy compression: Lowers size by deleting unnecessary information, and reducing the complexity of existing information. It has much higher compression ratios than lossless but with less quality of the file. **JPEG**, **MPEG**, and **MP3** use lossy compression algorithms.

However in this project, only lossless compression algorithms will be discussed. There are plenty of lossless compression algorithms, three will be mentioned here: **LZW**, **Huffman**, and **Arithmetic coding.**

PROBLEM DESCRIPTION:

Throughout this project, our problem is to design some compression algorithms without losing data. It is considered an optimization problem, all the information in the message must be reserved while the message is reduced to the least minimum size. We will use two methods to solve such a problem:
-Dictionary-based encoding: searches for matches between the message and the dictionary. When it finds a match, it substitutes a reference to its index in the dictionary.
-Entropy encoding: which is basically compressing the data by replacing each input symbol with ASCII fixed-length to a corresponding variable-length prefix codeword. The symbol that appears more in the message will be given the shortest codeword in order to achieve the reduced size of the message. Huffman, and Arithmetic are the most common techniques in entropy encoding.

In this project we will focus on encoding techniques(compression) rather than decoding techniques(decompression).

LOGICAL DESCRIPTION OF THE ALGORITHM:

-LZW(Lempel-Ziv-Welch) Coding:

LZW algorithm can be considered as a lazy quick algorithm that reads the data once and compresses as it goes through it. It mentions the data with the first occurrence it has, then saves it into its dictionary defined by an index if used in the future. The indexes that resemble compressed data in the dictionary are defined previously to distinguish between regular data and compressed data. This example will further explain LZW coding:

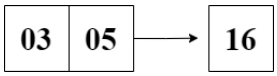
Data Range: 0 => 15 (14 bits)
The message that is given by index:

03	05	04	14	11	05	04	09	11	12	14	11	15	05	04	09	11	11	11

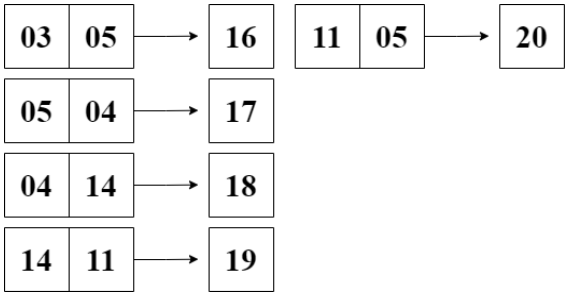
LZW coding will go through the data entirely and save it to its dictionary, here the range of data is 0 to 15. So,we will start our dictionary with an index of 16.

First: We start reading 03, and check if it is present in the dictionary. In this case, our dictionary is empty so we add it to the compressed data.
Compressed data: 03

Second: Check if 05 is in the dictionary, if not then it is added as compressed data. Also add 03 05 into the dictionary, if we ever see it later.
Compressed data: 03 05



We do so for the next couple of data without any saved dictionary indexes:



Compressed data: 03 05 04 14 11 05

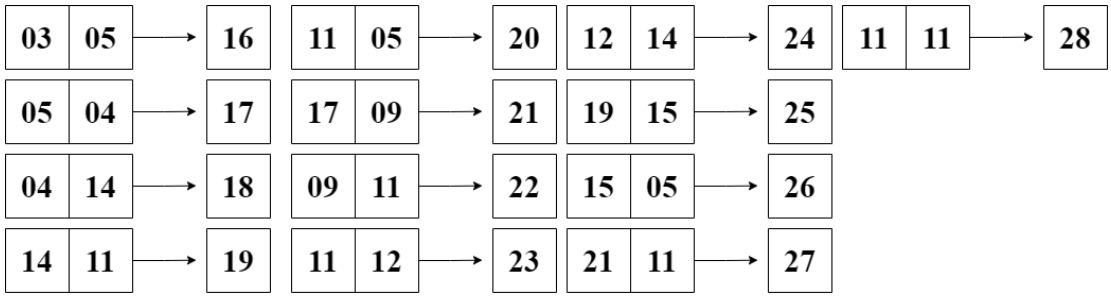
Third: reaching index 7 from our data with a value of 04. We check if 05 04 is present, and it is! So we add an index of 17 to our compressed data, and remove the 05.
Compressed data: 03 05 04 14 11 17

Fourth: We continue with the same pattern with the compressed data taken. We check for the pattern 17 09, in this case it will also be added to the dictionary.
Compressed data: 03 05 04 14 11 17 09

The same pattern is done for the whole data which results in a **compressed message** of:

03	05	04	14	11	17	09	11	12	19	15	21	11

Final Dictionary:

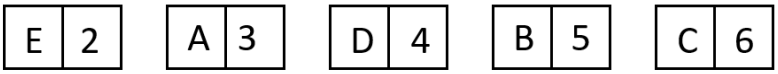


The LZW coding as mentioned, is a lazy algorithm that saves its dictionary as it goes for future situations going with the thought of “in case” it is present. The message started with 19 bytes, ending with 13 bytes with an overall reduction of **6 bytes**.

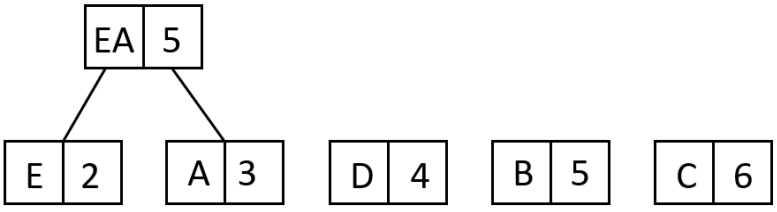
-Huffman Coding:

The idea of huffman encoding is to give a fewer number of bits or small codewords to the most frequent character in the message, and the long number of bits or large codewords to the least frequent character in the message. This assigning of codewords based on frequencies is done in order to achieve reduced size of the message. To understand the logic of huffman encoding better, let us go through an example; the message going to be compressed is:

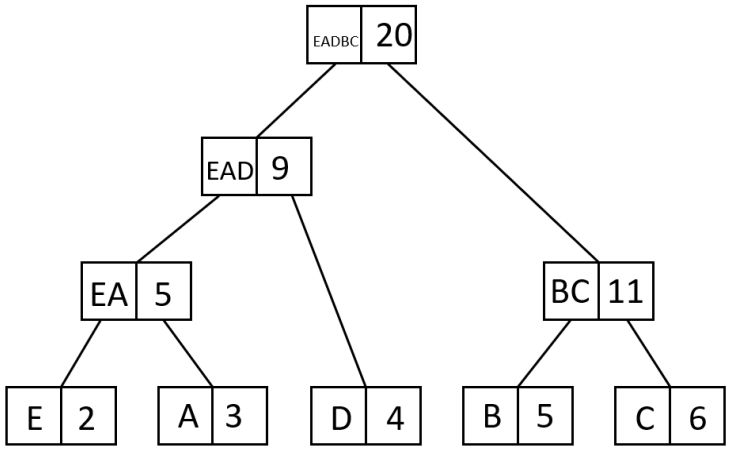
BCCABBDDAECCBBAEDDCC with size equals **8*20 = 160 bits**
First: Frequency table will be given as input.
Frequencies table: {A:3, B:5, C:6, D:4, E:2}
Second: we will implement a tree containing all characters sorted in increasing order based on frequency as leaf nodes.



Third: we are going to combine the least two characters in terms of their frequencies as a new node.

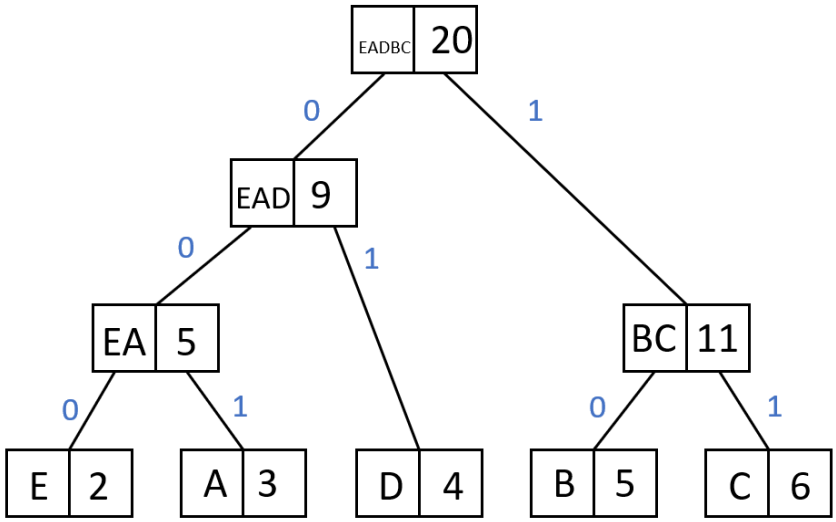


Fourth: repeat the third step until there are no two nodes to be combined.



Constructing the tree will help us define the code.

Fifth: We are going to mark the left hand edges as 0, and the right hand edges as 1.



Sixth: In order to define the codeword of a character, we are going to go through all the nodes starting from the root until reaching the character.

Codewords result:

A 001 **C** 11 **E** 000

B 10 **D** 01

These codewords now are used to encode the message in order to compress the message and reduce its size.

The new size for the message using huffman encoding will be:

(Frequency * #of codeword bits)

(3*3) + (5*2) + (6*2) + (4*2) + (2*3) = 45 bits

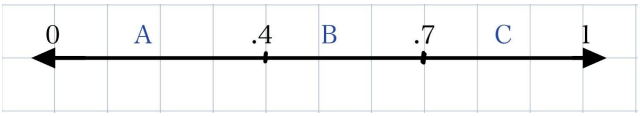
This is how Huffman encoding works, you can notice that the most frequent character is given a short number of bits **(character:C, codeword:11)** and the less frequent character is given the long number of bits **(character:E, codeword:000)**. Also, the size before using huffman encoding was **160 bits**, and with the help of huffman the message compressed and the size reduced to **45 bits** only. In brief, to encode using huffman encoding we are only using the codewords dictionary in order to replace each character by its corresponding codeword.

-Arithmetic Coding:

The idea of arithmetic encoding is to pick a decimal number between 0 and 1 based on the probability table, so actually a message and a probability table are given. How to calculate the probability table is not a concern of this algorithm, but it just gives a probability of how often we might see a character in a message. To illustrate the idea of this algorithm, an example is shown below:

Message: 'ABAC' **Probability table: {A:0.4, B:0.3, C:0.3}**

Starting with the number line, each character will take a portion based on its probability. **A** will take **40%** of the line and so on.



Now, we can start encoding, each character will have an interval, to calculate the interval these formulas are used:

Interval length = Probability(Char) * Range

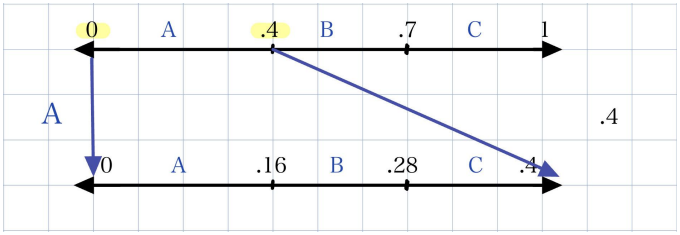
Min = Shift **Max = Min + Interval length**

The range of the number line now is 1-0 = 1. We will start with the first character **A**. Since **A** at the beginning of the number line, there's no shift here.

Probability(A)= .4 **Shift= 0** **Range= 1**

Interval length= .4*.1=.4 **Min= 0** **Max= 0+.4=.4**

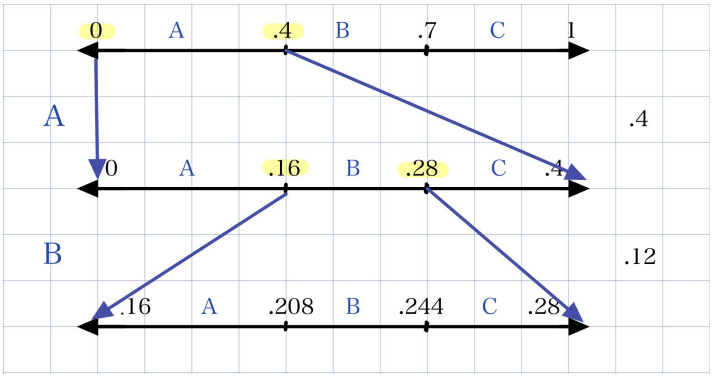
The new Range will be the interval of **A**. We will expand our number line.



In the new line we will do the same, we will divide it on the characters. Second character is **B**:

Probability(B)= .3 **Shift= .16** **Range= .4**

Interval length= .3*.4=.12 **Min= .16** **Max= .16+.12=.28**



The new Range is .12, we will do the same process for **A**:

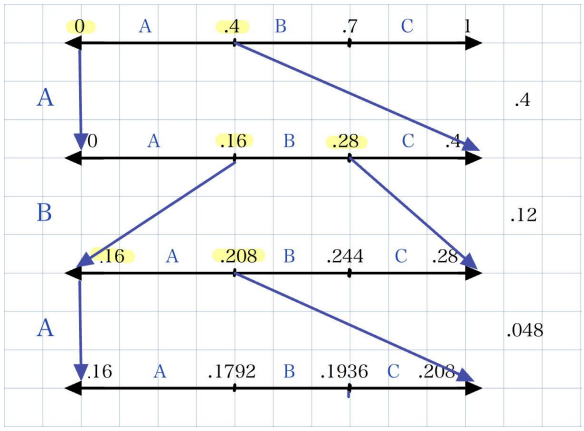
Probability(A)= .4 **Shift= .16** **Range= .12**

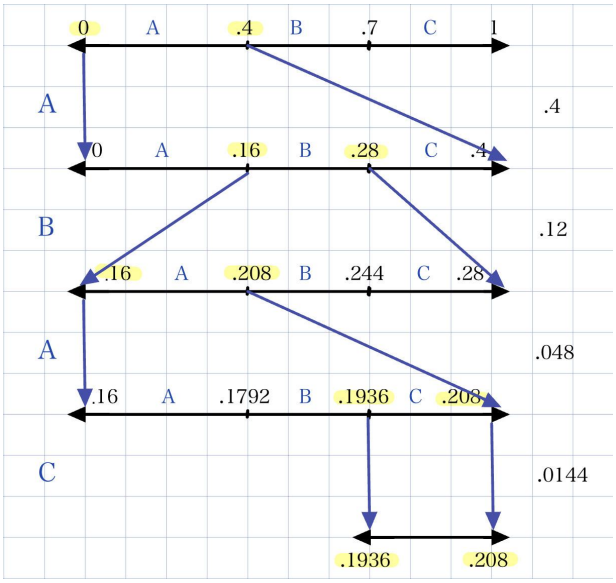
Interval length= .4*.12=.048 **Min= .16** **Max= .16+.048=.208**

Repeat for the last character C:

Probability(C)= .3 **Shift= .1936** **Range= .048**

Interval length= .3*.048 =.0144 **Min= .1936** **Max= .1936+.0144=.208**





We are done! So our interval is [.1936, .208]. You can pick any number of this interval, we will take its mean: .1936+.208 / 2 = .2008

'ABAC' -> .2008

Another example: **'AACABA' -> 0.120832** notice that the message size is 6 bytes(6 characters) while the decimal is 4 bytes![1](float precision)

[1] It depends on the implementation and the programming language

PSEUDOCODE:

-LZW:

```
Input: msg(string)
Output: result(string)
Algorithm LZW_encode(msg):
    w <- Φ
    result <- Φ
    dictionary <- dict of ASCII code and their indices
    i <- dictionary.length
    for c in msg:
        wc <- w + c
        if wc in dictionary:
            w <- wc
        else:
            result += dictionary[w]
            dictionary[wc] = i
            i += 1
            w <- c
    if w:
        result += dictionary[w]
    return result
```

-Huffman Coding:

```
Input: freq_table(dictionary)
Output: huffman tree(dictionary)
Algorithm huffman_dict(freq_table):
    nodes <- Φ
    for symbol in freq_table:
        nodes.heappush(node(freq_table[symbol], symbol))
    while nodes.length > 1:
        left <- nodes.heappop()
        right <- nodes.heappop()
        left.huff <- 0
        right.huff <- 1
        newNode <- node(left.freq + right.freq, left.symbol +
        right.symbol, left, right)
        heapq.heappush(newNode)
    return construct_dict(nodes[0])
```

```
-------------------------------------------------------------

Input: msg(string)
Output: encoded_msg(string)
    Algorithm huffman_encode(msg):
        encoded_msg <- Φ
        for c in msg:
            encoded_msg += codeword_dict[c]
        return encoded_msg
```

-Arithmetic Coding:

```
Input: msg(string), prob_table(dictionary)
Output: encoded decimal
Algorithm arithmetic_encode(msg, prob_table):
    min <- 0
    max <- 1
    r <- max - min
    for c in msg:
        min <- min + c_prob_table[c][2] * r
        r <- prob_table[c] * r
        max <- min + r
    return (max + min) /2
```

[2] Cumulative probability table is used here to calculate the shift.

IMPLEMENTATION:

Check out this [github respiratory] for the whole implementation.

ALGORITHM ANALYSIS:

Here's a table[3] that summarizes the differences between these algorithms

	LZW	**Huffman**	**Arithmetic**
Data Structure	The data structure used: **Dictionary (Hashmap)** LZW is based on pointers that are pointing to a specific sequence of strings that appears in its dictionary. The dictionary is first assigned for all ASCII codes, and by reading new codes are created that are a combination of more codes.	The data structure used here are: **-Dictionary** Which contains each character with its codeword, used in encoding and decoding the message. **-Minimum heap & Binary tree** are used in order to compare two nodes in terms of the least frequency value, and to find the minimum external path weight.	The only data structure used here is: **Dictionary(Hashmap).** There are two dictionary: **-prob_table:** (input) maps each char of the alphabets to its probability. **-c_prob_table:** (used for shifts) maps each char of the alphabets to its cumulative probability.
Algorithm Paradigm	**Greedy** LZW is designed to find the longest possible string before it makes a transmission. Adding to previously defined indexes and using the longest of the match.	**Greedy** In every stage it looks at the occurrence of each character and tries to find the free prefix binary code in order to minimize the expected codeword for compressing the data.	**Greedy** In each iteration it tries to calculate the optimal prefix(smallest decimal) for the sub-message till the whole message is encoded

[3] Everything mentioned in the table is concerned for the encoding process only(compression)

Correctness	100%	100%	100%
Performance[4]	Highest	Middle	Lowest
Efficiency[5]	Lowest	Middle	Highest
Uses	-GIF -PDF,TIFF -Unix's'compress' command	-PNG -ZIP -MP3	-HEVC -JPEG, MPEG -Deep Learning
Time Complexity[6]	**n** LZW's code loops in the ASCII dictionary $\in$ **O(1)** Then it loops through the entire message once $\in$ **O(n)** Overall runtime $\in$ **O(n)**	**Σlog(Σ) + n** Using the dictionary, the algorithm iterates over the message which takes **O(n)**, and since we are going through all the alphabet inside the tree as well as the heap finds the minimum each time which takes **O(Σlog(Σ))**. The overall runtime is **O(Σlog(Σ)+n)**	**Σ + n** The algorithm iterates once over the message, all the operations inside the loop are constant so $\in$ **O(n)** but to construct the c_prob_table, we need to loop over the alphabet so $\in$ **O(Σ)** Overall runtime $\in$ **O(Σ + n)**

[4] In terms of execution time.
[5] In terms of compression ratio: original size:compressed size
[6] **Σ** is the alphabet size, **n** is message length

EXPERIMENTAL RUNNING TIME:

-Experimental computer specs:

CPU	RAM	System Type	OS
Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30GHz	16.0 GB	64-bit operating system, x64-based processor	Windows 11 Home

-Running times for each algorithm:

	LZW	Huffman	Arithmetic
Σ = 38 & **n** = 500 (numbers & punctuations)	1.0004044e-4s	1.0020733e-4s	9.0043545e-4s
Σ = 52 & **n** = 1000 (letters)	9.946823e-4s	9.629726e-4s	2.1998167e-2s
Σ = 90 & **n** = 5000 (both)	3.0374527e-3s	1.9934177e-3s	1.652555466e-1s

-Input size versus running time diagrams:

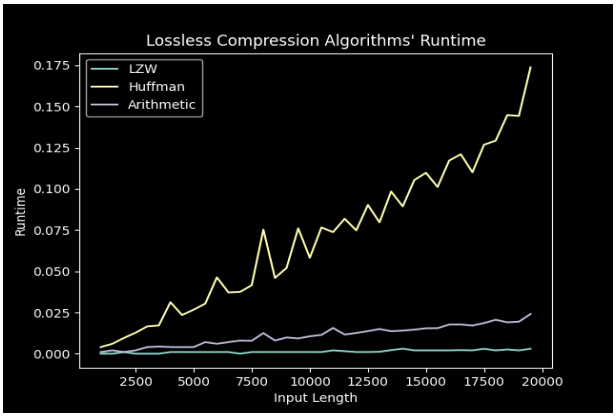


Figure1: N is fixed, Σ is variable

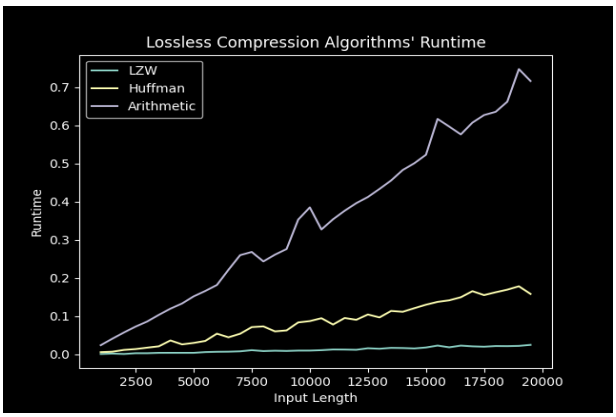


Figure2: Σ is fixed, N is variable

As you can see in Figure1, N is fixed. When Σ increases, Huffman runtime increases notably, because Huffman deals with Σ in Σlog(Σ) time, while Arithmetic and LZW are linear.
In Figure2, Σ is fixed, so all algorithms are asymptotically equivalent, but you can see that Arithmetic is increasing rapidly, and that's due to the fact that Arithmetic has many arithmetic operations which scales up the running time notably.

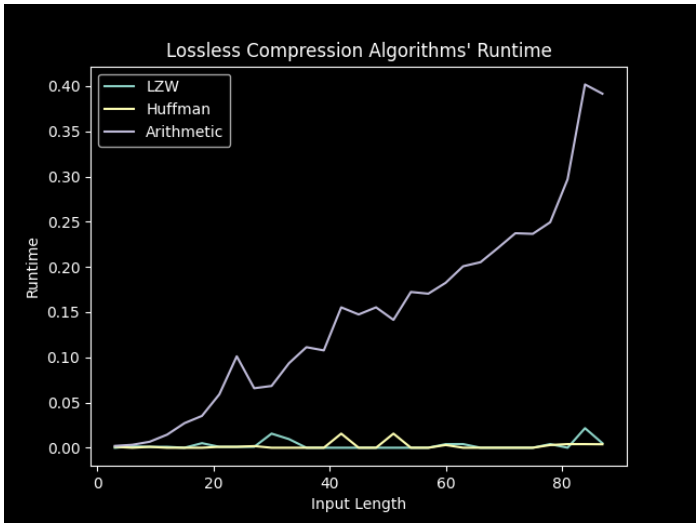


Figure3: N is variable, Σ is variable

In Figure3, both N and Σ are changing, though huffman has the greatest time complexity, arithmetic has the greatest execution time. LZW is the fastest.

CONCLUSION:

This project discusses optimization problems by introducing three types of greedy lossless compression algorithms that are aimed to reduce the size of a message by reserving the information. It explains its algorithms, implementation, algorithm analysis, and run time analysis. Our results show that Arithmetic algorithms are used for high efficiency with no regards to time, and LZW for vice versa. With that being said, Huffman is the middle ground between both that is known to be widely used. Compression algorithms are one of the most important algorithms that needs to be studied due to the high importance of minimizing the space and time that results in significant cost saving.

REFERENCES:

- *Barracuda Networks*. What is Data Compression? | Barracuda Networks. (n.d.). Retrieved November 28, 2022, from https://www.barracuda.com/glossary/data-compression#:~:text=Data%20compression%20is%20the%20process,bits%20than%20the%20original%20representation
- Gad, A. (2022, November 14). *Lossless data compression using arithmetic encoding in python and its applications in Deep Learning*. neptune.ai. Retrieved November 28, 2022, from https://neptune.ai/blog/lossless-data-compression-using-arithmetic-encoding-in-python-and-its-applications-in-deep-learning
- *Huffman coding: Greedy Algo-3*. GeeksforGeeks. (2022, October 26). Retrieved December 7, 2022, from https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/
- *3.4 huffman coding - greedy method*. YouTube. (2018, February 8). Retrieved December 7, 2022, from https://youtu.be/co4_ahEDCho
- Rosetta Code. (2022, October 28). LZW compression. Rosetta Code. Retrieved December 7, 2022, from https://rosettacode.org/wiki/LZW_compression
- YouTube. (2021, February 17). *LZW compression algorithm explained | an introduction to data compression*. YouTube. Retrieved December 7, 2022, from https://www.youtube.com/watch?v=KJBZyPPTwo0&t=66s
- *LZW Lempel Ziv Compression Technique*.(2021, 08 Nov). Retrieved December 8, 2022, from https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/
- Shahbahrami, A., Bahrampour, R., Sabbaghi Rostami, M., & Ayoubi Mobarhan, M. (n.d.). *Evaluation of huffman and arithmetic algorithms for multimedia compression standards*. NASA/ADS. Retrieved December 9, 2022, from https://ui.adsabs.harvard.edu/abs/2011arXiv1109.0216S/abstract
- *Analysis of huffman coding and Lempel Ziv Welch (LZW) coding ... - ISROSET*. (n.d.). Retrieved December 9, 2022, from https://www.isroset.org/pub_paper/IJSRCSE/5-ISROSET-IJSRCSE-02950.pdf
- Wiseman,Y.(2007, 5 January) *The relative efficiency of data compression by LZW and LZSS*. Retrieved December 12, 2022, from https://www.jstage.jst.go.jp/article/dsj/6/0/6_0_1/_pdf/-char/ja