LOSSLESS DATA COMPRESSION ALGORITHMS AND THEIR COMPARISON

Project Report

*submitted in partial fulfillment of
the requirements for the award of the degree of*

**BACHELOR OF TECHNOLOGY
in
COMPUTER SCIENCE Specialization in
Cloud Computing and Virtualization Technology**

By :

NAME	**ENROLLMENT NO.**
ASHI AGARWAL	R110216043
DEEPANSHU GOYAL	R110216057
ARPIT BHARDWAJ	R110216039
ASHISH BANSAL	R110216044

Under the guidance of

MR. G.L. PRAKASH
Assistant Professor, Department of Virtualization

UPES
UNIVERSITY WITH A PURPOSE

**Department of Virtualization
School of Computer Science**

UNIVERSITY OF PETROLEUM AND ENERGY STUDIES Bidholi, Via Prem Nagar, Dehradun, Uttarakhand
2018-19

UPES
UNIVERSITY WITH A PURPOSE

CANDIDATES DECLARATION

I/We hereby certify that the project work entitled **Lossless Data Compression Algorithms and their Comparison** is in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology in Computer Science And Engineering with Specialization in Cloud Computing and Virtualization Technology and submitted to the Department of Virtualization at School of Computer Science, University of Petroleum And Energy Studies, Dehradun, is an authentic record of my/our work carried out during a period from **September, 2018** to **December, 2018** under the supervision of **Mr. G.L. PRAKASH, Assistant Professor, Department of Virtualization**

The matter presented in this project has not been submitted by me/ us for the award of any other degree of this or any other University.

Ashi Agarwal
Roll No R110216043
Deepanshu Goyal
Roll No R110216057
Arpit Bhardwaj
Roll No R110216039
Ashish Bansal
Roll No R110216044

This is to certify that the above statement made by the candidate is correct to the best of my/our knowledge.

Dr. Amit Agarwal
Head Department of Virtualization
School of Computer Science.
University of Petroleum And Energy Studies Dehradun - 248001
(Uttarakhand)

Mr. G.L. Prakash
Project Guide
Asst. Professor(SG)
School of Computer Science,
UPES, Dehradun

ACKNOWLEDGEMENT

We wish to express our deep gratitude to our guide **Mr. G.L. Prakash**, for all advice, encouragement and constant support he has given us throughout our project work. This work would not have been possible without his support and valuable suggestions.

We sincerely thank to our Head of the Department, **Dr. Amit Agrawal**, for his great support in doing our project **Lossless Data Compression Algorithms and their Comparison** at SoCS.

We are also grateful to **Dr. Manish Prateek Professor and Director SoCS** and **Dr. Kamal Bansal Dean CoES,** UPES for giving us the necessary facilities to carry out our project work successfully.

We would like to thank all our **friends** for their help and constructive criticism during our project work. Finally we have no words to express our sincere gratitude to our **parents** who have shown us this world and for every support they have given us.

Name	**Ashi Agarwal**	**Deepanshu Goyal**	**Arpit Bhardwaj**	**Ashish Bansal**
Roll No.	**R110216043**	**R110216057**	**R110216039**	**R110216044**

ABSTRACT

In today's world, With the advent of the Internet and mobile devices with limited resources and with the growing requirements of information storage and data transfer, Cloud Computing has become an important aspect, but cloud computing also require physical infrastructure, somewhere down the lane. This exponential sub purge of data leads to high demand for data processing that leads to a high computational requirement which is usually not available at the user's end. Compression reduces the redundancy in data representation thus increasing effective data density. [1] Data Compression is a technique which is used to decrease the size of data. This is very useful when some huge files have to be transferred over networks or being stored on a data storage device and the size is more than the capacity of the data storage or would consume so much bandwidth for transmission in a network. With the limited physical infrastructure for storage, data compression has gained even more importance these days. There are number of data compression algorithms, which are dedicated to compressing different data formats. Even for a single data type, there are number of different compression algorithms, which use different approaches. In this project, we will examine lossless data compression algorithms like Huffman encoding algorithm, Lempel-Ziv-Welch algorithm, and Shannon-Fano algorithm and comparing their performance. [2]

Keywords: Cloud Computing, Data Compression, Huffman encoding algorithm, Lempel-Ziv-Welch algorithm, Shannon-Fano algorithm.

TABLE OF CONTENTS

Contents

LIST OF TABLES

LIST OF FIGURES

Chapter 1. INTRODUCTION

Compression is the art of representing the information in a compact form rather than its original or in uncompressed form [3]. In other words, using the data compression, the size of a particular file can be reduced. This is very useful when processing, storing or transferring a huge file, which needs lots of resources. If the algorithms used to encrypt works properly, there should be a significant difference between the original file and
the compressed file. When data compression is used in a data transmission application, speed is the primary goal. Speed of transmission depends upon the number of bits sent, the time required for the encoder to generate the coded message and the time required for the decoder to recover the original message. In a data storage application, the degree of compression is the primary concern. There are two types of data compressions ie. Lossless data compression and Lossy data compression.

Table 1.0 :Comparison between Lossy and Lossless Data Compression Technique.

Lossless data compression	Lossy data compression
In Lossless data compression algorithms, the original data can be recovered from compressed data after applying decompression algorithm	In Lossy compression algorithms it permanently reduces the original data by eliminating certain information, especially redundant information, after decompressing the file only the part of original data is recovered.
Lossless compression is generally used for text data or spreadsheet files, where even a very small amount of data loss can be detected by users.	Lossy compression is generally used for video and sound, where a certain amount of information loss will not be detected by most users.
No loss in information so compression rate is small.	In return for accepting this distortion in reconstructed data we obtain high compression rate.
Less data can be accommodated in channel.	More data can be accommodated in channel.
E.g(*i*)*Fax Machine*,(*ii*)*Radiological Imaging*	*E.g.*(*i*)*Telephone System*,(*ii*)*Video CD*

Various lossless data compression algorithms have been proposed and used. Some of the main techniques in use are the Huffman Coding, Run Length Encoding, Arithmetic Encoding and

Dictionary Based Encoding [4]. We will examine the performance of the Huffman Encoding Algorithm, Shannon Fano Algorithm, and Lempel Zev Welch Algorithm. In particular, performance of these algorithms in compressing text data will be evaluated and compared.

Shannon-Fano Algorithm:

In Shannon–Fano Algorithm, the symbols are arranged in order from most probable to least probable, and then divided into two sets whose total probabilities are as close as possible to being equal. All symbols then have the first digits of their codes assigned; symbols in the first set receive "0" and symbols in the second set receive "1". As long as any sets with more than one member remain, the same process is repeated on those sets, to determine successive digits of their codes. When a set has been reduced to one symbol, of course, this means the symbol's code is complete and will not form the prefix of any other symbol's code. The algorithm works, and it produces fairly efficient variable-length encodings; when the two smaller sets produced by a partitioning are in fact of equal probability, the one bit of information used to distinguish them is used most efficiently. Unfortunately, Shannon–Fano does not always produce optimal prefix codes. [5]

Huffman Encoding:

The Huffman encoding works on variable length encoding rather than fixed length encoding. frequency of every character is calculated, and the lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code. The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream for this task a binary tree is created using the symbols as leaves according to their probabilities and paths of those are taken as the code words.

The Lempel-Ziv 77 Algorithm:

It is a dictionary-based compression algorithm As in Dictionary the set of all possible word of a language is stored similarly in LZ77 algorithm a dictionary is used to store or index the previously used string patterns In the compression process those index value is used instead of repeating. The dictionary is created dynamically in the compression process and no need to

transfer it with the encoded message for decompressing. In the decompression process, the same dictionary is created dynamically. Therefore, this algorithm is an adaptive compression algorithm.
Performance of compression algorithm [6] is based on space efficiency and the time complexity.The compression behaviour of algorithm is dependent on redundancy of symbol in source file therefore it is difficult to measure the performance of compression algorithm. There are some following measurements used to evaluate the performances of compression algorithms.

Compression Ratio –Compression Ratio is the ratio between the size of the compressed file and the size of the source file
Compression ratio= (size of compressed file)/ (size of source file)

Compression Factor-Compression factor is inverse of Compression ratio.It tells how much times our file has been compressed
Compression Factor= (size of source file)/ (size of compressed file)

Saving Percentage-It tells the shrinkage of the source file in percentage
Saving Percentage= (size before compression-size after compression)/ (size before compression) %

Time complexity –The time complexity is measured by the number of clocks used to encode the source code. The algorithm that uses less clock cycle to encode is considered more efficient.

In this project we will be comparing all the three algorithms on the basis of these factors mentioned above.

Chapter 2.

LITERATURE REVIEW

Data Compression is the way that you can use the space on cloud i.e. Server in an optimal way. In this project we will Lossless Data Compression algorithms which can reconstruct the original message exactly from the compressed message Here is the conclusion of some of the reference paper that we review to make our project better and to know more technologies that we can use in our system.

In the paper [6] by S.R. KODITUWAKKU, Department of Statistics & Computer Science, University of Peradeniya, Sri Lanka, U. S. AMARASINGHE, Postgraduate Institute of Science, University of Peradeniya, Sri Lanka; Among the available lossless compression algorithms they considered the Run Length Encoding Algorithm, Huffman Encoding, The Shannon Fano Algorithm, Arithmetic Encoding, The Lempel Ziv Welch Algorithm for study. They carried out an experimental comparison of a number of different lossless compression algorithms for text data. On the basis of compression times, decompression times and saving percentages of all the algorithms, they found that the Shannon Fano algorithm can be considered as the most efficient algorithm among the selected ones. The values which they calculated are in the acceptable range and it also shows better results for the larger files.

In the paper [7] by Laxmi Shaw, Student Member, IEEE, Daleef Rahman, and Aurobinda Routray, Senior Member, IEEE; The authors have examined the different lossless compression methods for single and multichannel EEG signals, and their performance with respect to their relative Compression ratios has been analysed. They evaluate their proposed algorithms, analysis of which showed that a very high CR (Compression Ratio) in different publicly available database. They also analysed that among the existing methods for the single-channel EEG compression scheme, the linear prediction followed by the context-based error modelling showed the best results. The increase in CR by applying the context-based error modelling is high for the first-order predictor, whereas the increase is small for higher-order predictors. The MVAR model and the bivariate autoregression model were examined for the multichannel EEG compression. The results show that these proposed methods in combination outperform the existing MVAR and the bivariate autoregressive model.

Chapter 3
PROBLEM STATEMENT

Storage on the cloud is a limited resource. Even though more storage space can be purchased, it seems better to utilize the given space to the fullest. The solution to this problem is data compression. Compress data to save space and then store it on the cloud, also in doing so, we save the data transmission cost over the network and make our cloud storage even more efficient.

OBJECTIVES

- This project will compare and contrast different lossless data compression algorithm theoretically and practically.
- Algorithm to be discussed in this project are:
  huffman encoding
  shannon fano algorithm
  lempel-ziv 77 algorithm
- This project will compare the algorithms on the basis of data to be compressed and then tabulate the obtained statistics.
- Further objectives may involve the discussion on optimization and deployment on cloud(if possible,time constraint).

Chapter 4.
METHODOLOGY

· Research .
· Analysis.
· Selection of Text file to be Compressed.
· Compress using all three Data Compression algorithms:
    1. Huffman Encoding Algorithm.
    2. Shannon-fano Algorithm.
    3. Lempel-ziv 77 Algorithm.

·Calculate all comparison factors for all three algorithms.
    1. Compression Ratio.
    2. Compression Factor.
    3. Time Complexity.
    4. Encoded File Size.
    5. Saving Percentage.

· Comparison between algorithms on the basis of above factors.
· Showing Results in tabular form

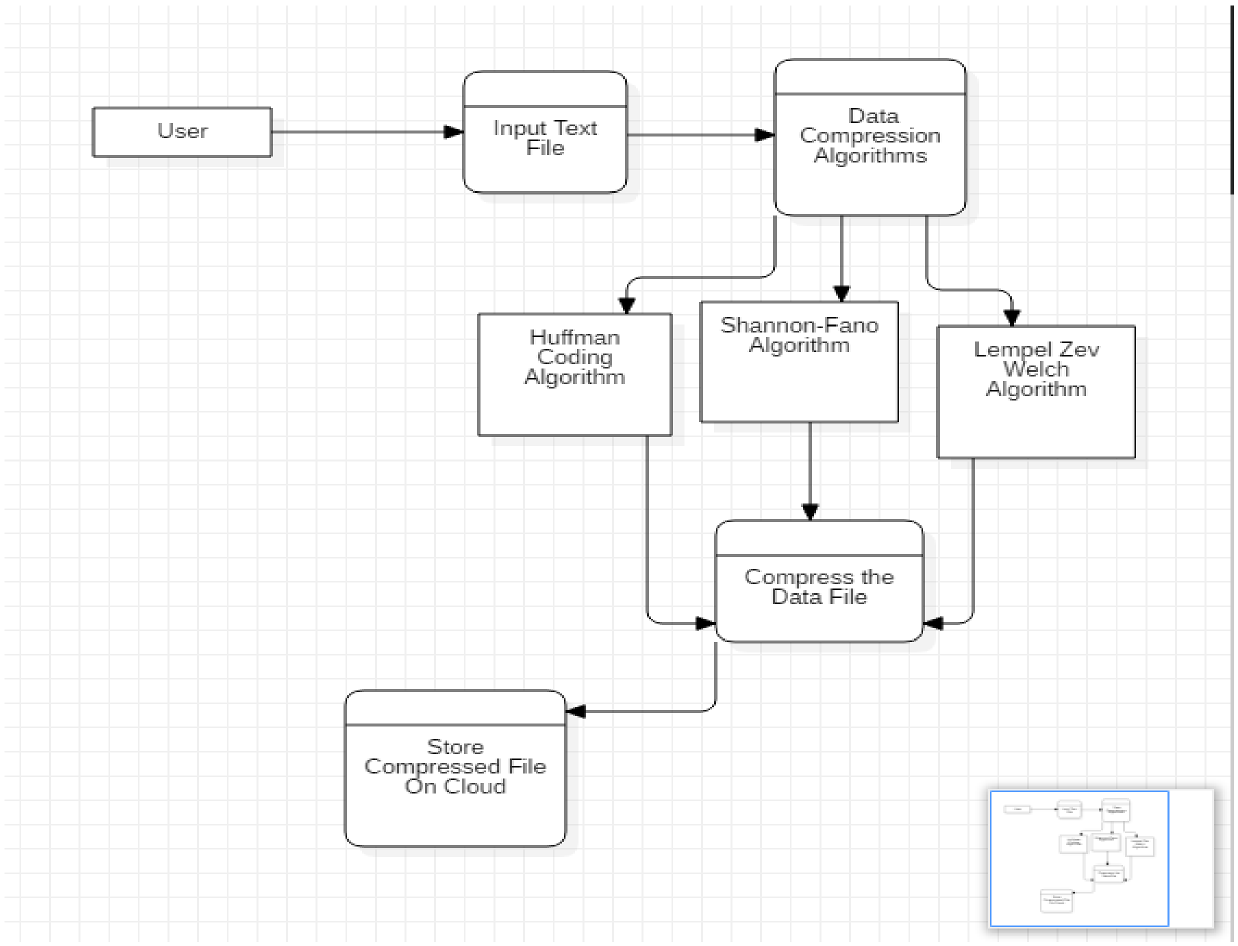

Figure 1.1:Methodology of Project

Chapter 5
ALGORITHM

Data Compression will be done by these of three Algorithms defined below:

Huffman Coding:

Huffman coding is lossless data compression algorithm that comes under Greedy Algorithm. The basic idea behind Algorithm is to assign variable length code to each character based on their frequency
There are two part of Huffman encoding
a) Creating a Huffman tree
b) Traverse the tree and assign code to each character

a. Creating a Huffman tree

Step1- Create a leaf node for each unique character and build a min heap for each character
Step2- Extract two node (Whose frequency is minimal) from min heap say node1 and node2
Step3- Create a new internal node whose frequency is equal to sum of two extracted node assign node 1 as a left child and node 2 as aright child
Step4-Repeat step2 and step3 until the min heap does not contain only one node. The remaining node is root node and the tree has been completed

b. Traverse the tree and assign code to each character

Step1- Traverse the tree formed starting from the root. Maintain an auxiliary array.
Step2- While moving to the left child, write 0 to the array.
Step3- While moving to the right child, write 1 to the array.
Step4- Print the array when a leaf node is encountered.

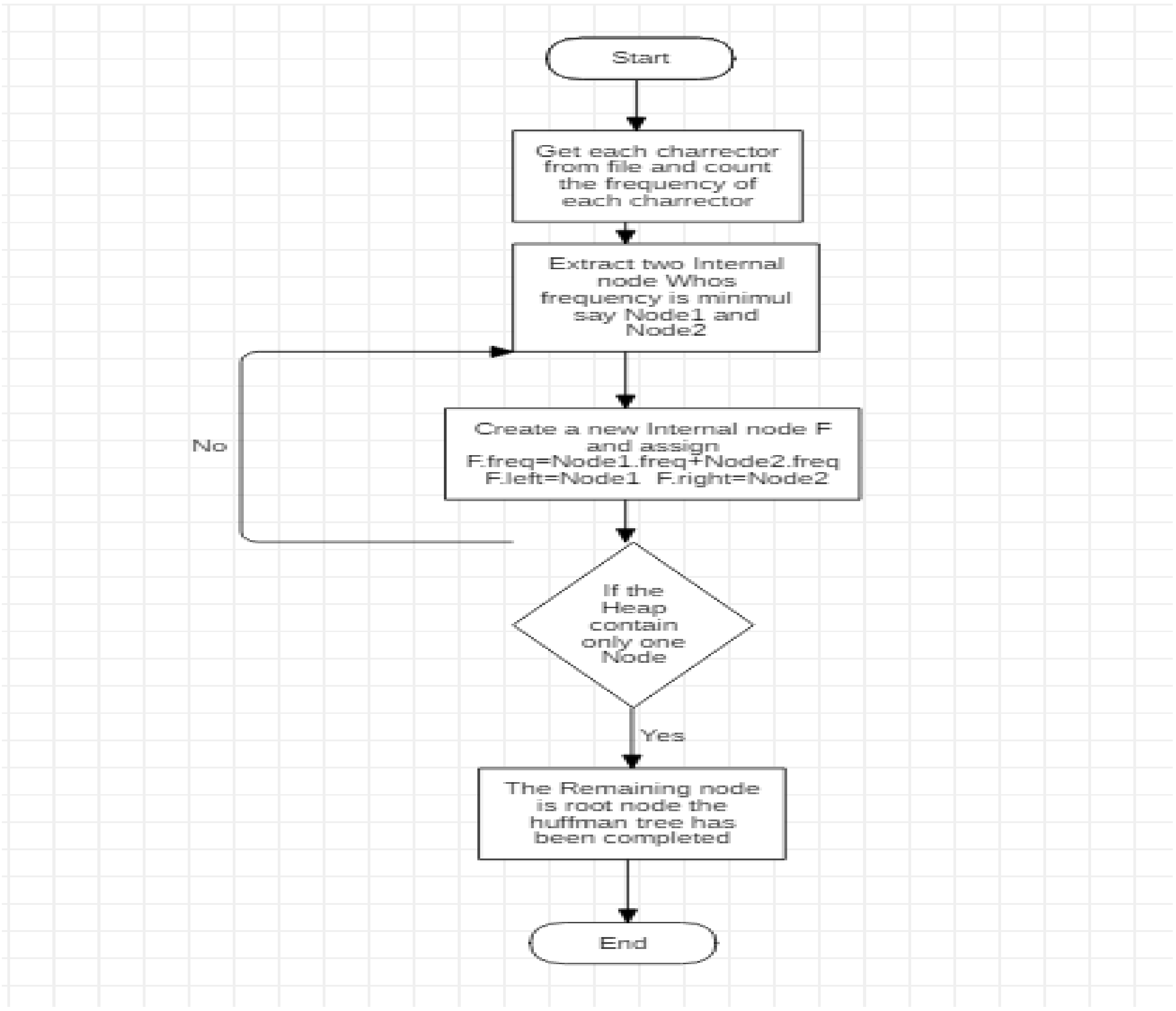
Start
Get each charrector from file and count the frequency of each charrector
Extract two Internal node Whos frequency is minimul say Node1 and Node2
Create a new Internal node F and assign F.freq=Node1.freq+Node2.freq F.left=Node1 F.right=Node2
No
If the Heap contain only one Node
Yes
The Remaining node is root node the huffman tree has been completed
End

Figure 1.2: Huffman Encoding Flowchart

Shannon-Fano :

Shannon-fano algorithm is a lossless data compression algorithm named after Claude Shannon and Robert Fano, in this technique we construct a code for each character which depends upon the frequency of that character and then encode the file accordingly.
There are two part of Huffman encoding
a) Creating a Shannon-Fano tree
b) Traverse the tree and assign code to each character.
Creating a Shannon-Fano tree

Step 1: Calculate frequency of each unique character.
Step 2: Calculate probability of each unique character in the file.
Step 3: Arrange the probability in decreasing order.
Step 4: Divide the Array in two parts in such a way that sum of probability of one part almost equal to sum of probability of other part.
Step 5: Keep diving the array till only two probability are left in sub-parts.

Step 6: Assign 0 to all left sub-edges and 1 to all right sub-edges .
Traverse the tree and assign code to each character.
Step 1:Calculate the code for each character by taking 0 and 1 values of the edges required to reach the leaf node of that character.
Step 2:Traverse the file and write corresponding code of each character in a binary file.
Step 3:Save Binary file.

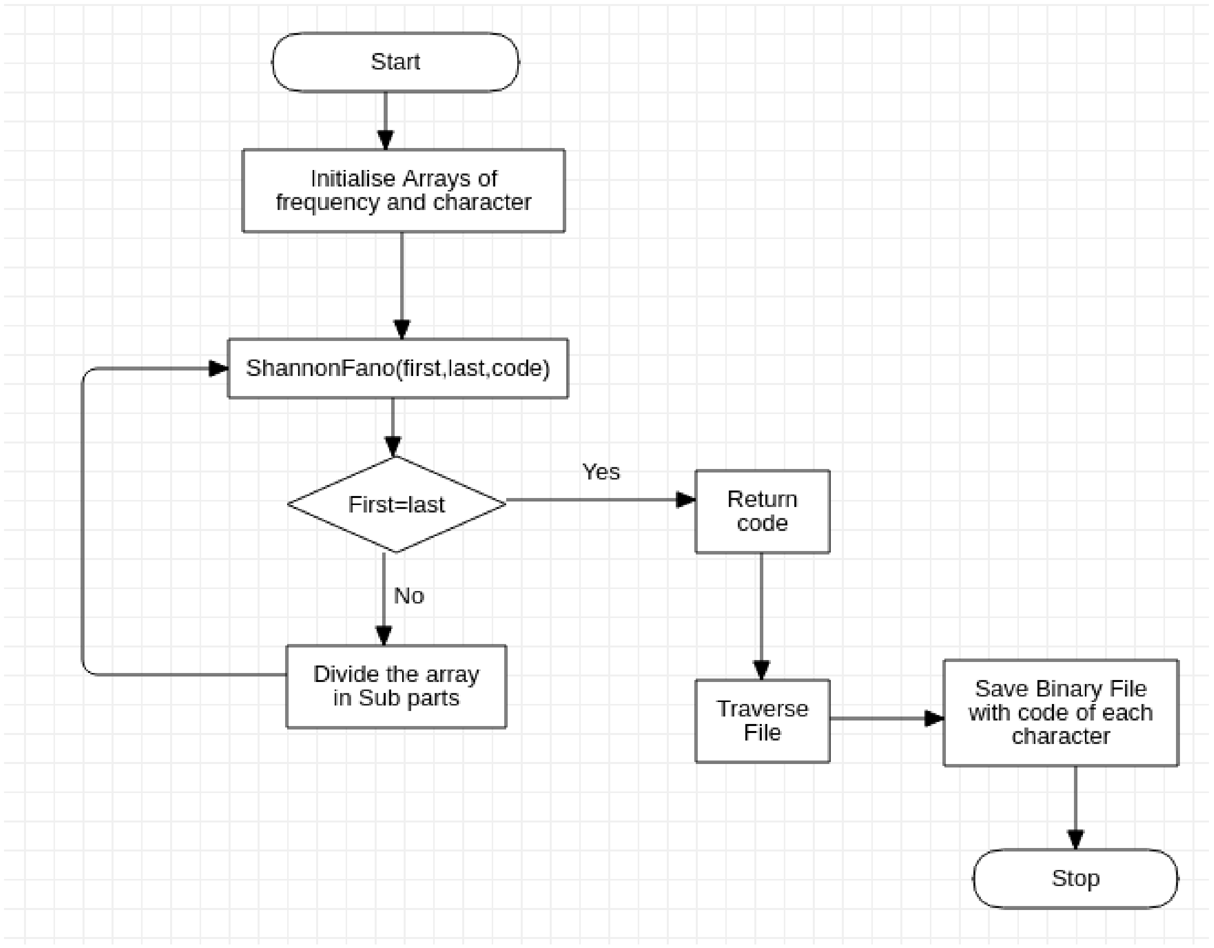
Start
Initialise Arrays of frequency and character
ShannonFano(first,last,code)
First=last
Yes
Return code
No
Divide the array in Sub parts
Traverse File
Save Binary File with code of each character
Stop

Figure 1.3: Shannonfano Flowchart

Lempel-Ziv-77 :

```
while (lookAheadBuffer not empty) {
get a reference (position, length) to longest match;
if (length > 0) {
output (position, length, next symbol);
shift the window length+1 positions along;
} else {
output (0, 0, first symbol in the lookahead buffer);
shift the window 1 character along;
}}
```

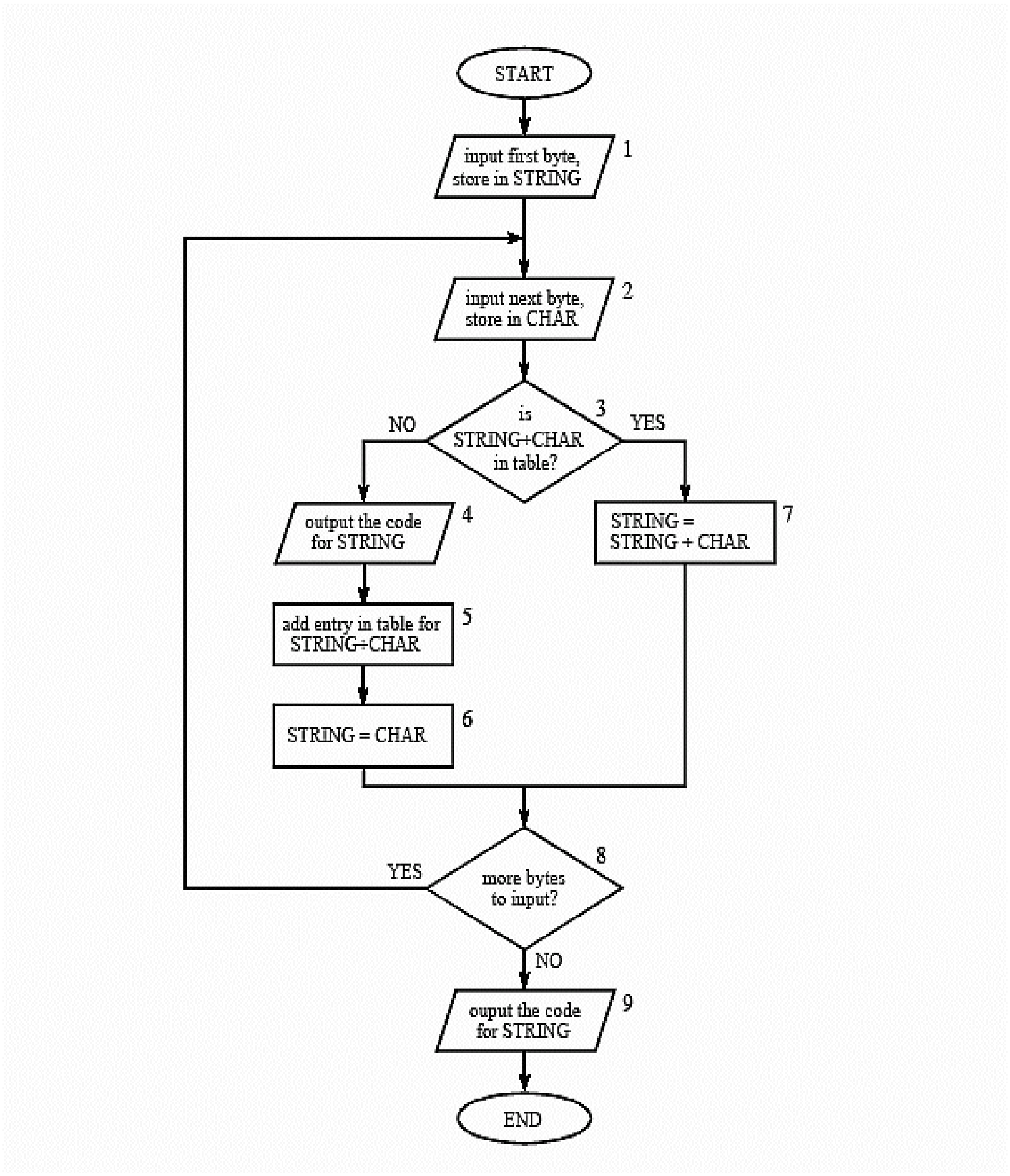
START
input first byte, store in STRING
1
input next byte, store in CHAR
2
is STRING+CHAR in table?
3
NO
YES
output the code for STRING
4
STRING = STRING + CHAR
7
add entry in table for STRING+CHAR
5
STRING = CHAR
6
more bytes to input?
8
YES
NO
ouput the code for STRING
9
END

Figure 1.4: Lempel-ziv 77 Flowchart

Chapter 6
SYSTEM REQUIREMENTS

- HARDWARE:
    1. 64 bits processor architecture supported by windows.
    2. Minimum RAM requirement for proper functioning is 8 GB.
    3. Required input as well as output devices.
- SOFTWARE:
    1. C Compiler (GCC).
    2. Git

Chapter 7
IMPLEMENTATION

Pseudo code for Huffman encoding:

Let C is set of n character such that every character c belongs to C every character has a attribute is frequency. And Q is a min priority queue based on frequency of characters

```
Huffman(C)
{
// input : C set of n character with their frequency
// Output : Huffman tree
  n=|C|
  Q=C
  for i=1 to n-1 do
  Allocate a new internal node z
  z.left=x=Extract-Min(Q)          //Extracting two internal nodes with lowest frequency
  z.right=y=Extract-Min(Q)         //And assigning that sum to new internal node
  z.freq=x.freq+y.freq
  Insert(Q,z)
  return Extract-Min(Q)  //return the root of the tree
}
```

Time complexity of creating Huffman tree

= (time complexity to Extract min node)*2(n-1) // where n is number of character

=O(logn)*2n

=O(nlogn)

Pseudo code for Shannon-Fano encoding:

Take two Arrays Probability[] which consist probability of each character ,and Character[] that consist the consist of all the character in the same order of there probability.

```
Probability[]={0.30,0.25,0.15,0.12,0.10,0.08}
Character[]={'a','c','s','b','m','n'}
Shannonfano( initial, final, code)
{
  If initial=final:
    print code and return.
  Else
        Assign first=initial,last=final
        Assign sumFront=probability[first], sumLast=probability[last]
        while first!=last:
                If sumFront>sumLast:
                    Decrement last by 1
                 endif
                    Assign sumLast=sumLast+probability[last]
                Else
                    Increment first by 1
                    Assign sumFront=sumFront+probability[first]
```

```
            endElse
      Assign code=code*10
     Shannonfano(initial,first,code+0)
     Shannonfano(first+1,final,code+1)
}
```

Time-Complexity of Shannon-Fano algorithm

$T(n) = T(n/2) + T(n/2) + n$

$T(n) = 2T(n/2) + n$

$\quad = O(n\text{Log}_2(n))$

Pseudo code for Lempel-ziv 77 algorithm:

```
{
Initialize table with single character strings
P = first input character
WHILE not end of input stream
C = next input character
IF P + C is in the string table
P = P + C
ELSE
output the code for P
add P + C to the string table
P = C
END WHILE
output code for P
}
```

Chapter 8
SCHEDULE (PERT CHART)

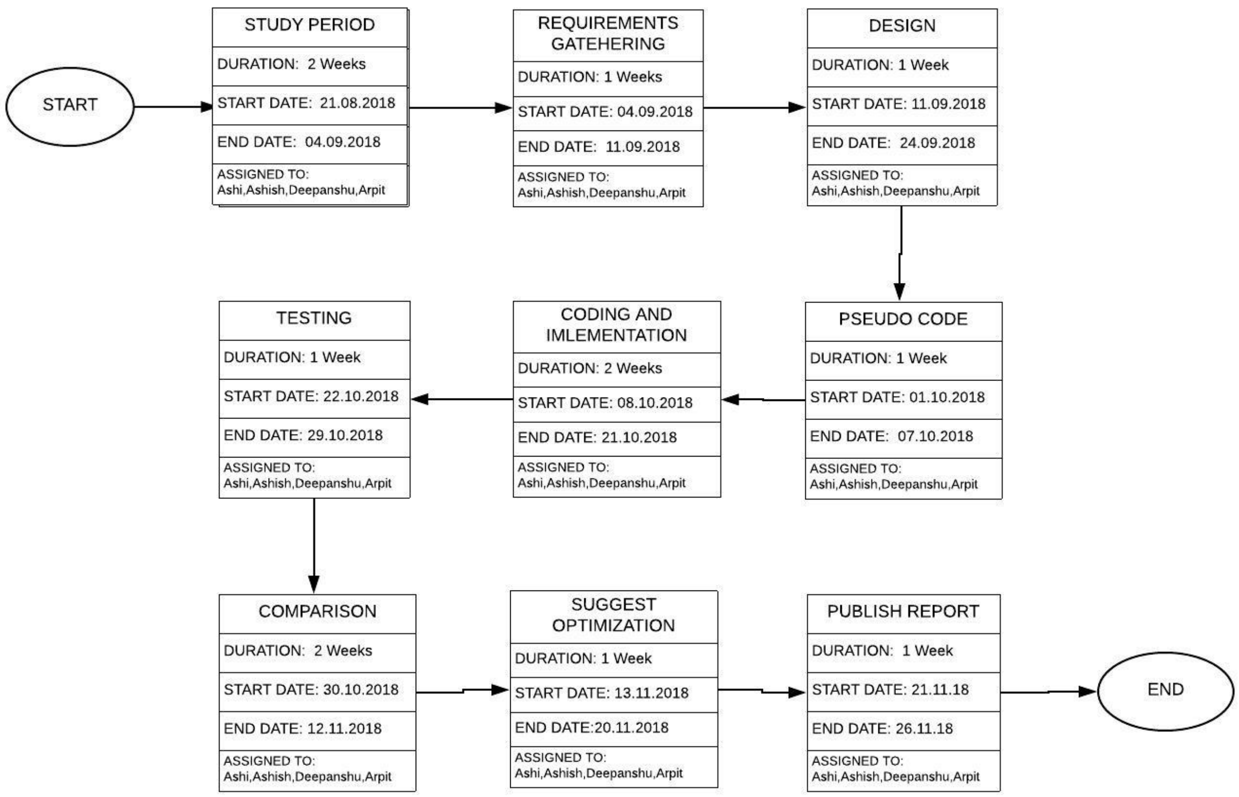

Figure 1.5: Pert chart

Chapter 9
LIMITATIONS AND FUTURE SCOPE

1. These three algorithms are conventional algorithm and enough modification and optimization has already been done so further optimization can't be suggested.
2. Compression of different Characters other than 128 and symbol which are not defined in the dictionary,can't be achieved.
3. Compression of databases which contains less redundant data is not computationally profitable in lempel ziv algorithm.
4. In future we will also be implementing decompressor for all three algorithms.
5. Storing file automatically on cloud by simply running a command.

Chapter 10
RESULTS AND ANALYSIS

Input file size : 2272 bytes

File name : input.txt

Table 1.1-Uniform distribution

Comparison Factors	**Huffman encoding algorithm**	**Shannon fano algorithm**	**Lempel-ziv 77 algorithm**
Compression ratio	0.25080	0.481954	0.577465
Compression Factor	3.985965	2.074886	1.731707
Encoded File size	570 bytes	1095 bytes	1312 bytes
Time complexity	0.000667	0.001052	0.001119
Length of code	271	116	170
Saving percentage	74.911972	51.804577	42.253521

Conclusion for table 1.1

For a uniformly distributed data in a file the compression ratio for the huffman encoding algorithm is least, that is for a uniformly distributed file Huffman encoding algorithm is working best and saving percentage is also high with minimum time complexity also reduces the file size by 75 % (Approx) While Shannonfano reduces it to 52% (Approx) and LZ77 reduces it to 42% (Approx).

Input file size : 1384 bytes

File name : input3.txt

Table 1.2-Non Uniform(even) distribution

Comparison Factors	**Huffman encoding algorithm**	**Shannon fano algorithm**	**Lempel-ziv 77 algorithm**
Compression ratio	0.251445	0.357659	0.531792
Compression Factor	3.977011	2.795960	1.880435
Encoded File size	348 bytes	495 bytes	736 bytes
Time complexity	0.000526	0.000605	0.000904
Length of code	271	116	170
Saving percentage	74.855492	64.234100	46.820812

Conclusion for table 1.2

For a Non-uniform evenly distributed data in a file the compression ratio for the huffman encoding algorithm is least, that is for a Non-uniform evenly distributed file Huffman encoding algorithm is working best and saving percentage is also high with minimum time complexity also reduces the file size by 75% (Approx) While Shannonfano reduces it to 64% (Approx) and LZ77 reduces it to 46% (Approx).

Input file size : 4840 bytes

File name : input1.txt

<u>Table 1.3-Non Uniform distribution</u>

Comparison Factors	**Huffman encoding algorithm**	**Shannon fano algorithm**	**Lempel-ziv 77 algorithm**
Compression ratio	0.249174	0.429339	0.119008
Compression Factor	4.013267	2.329163	8.402778
Encoded File size	1206 bytes	2078 bytes	576 bytes
Time complexity	0.001453	0.001420	0.005355
Length of code	271	116	170
Saving percentage	75.082642	57.066113	88.099174

Conclusion for table 1.3

For a Non-uniformly distributed data in a file the compression ratio for the LZ77 algorithm is least, that is for a Non-uniformly distributed file LZ77 algorithm is working best and saving percentage is also high also reduces the file size by 88% (Approx) ,While Shannonfano reduces it to 57% (Approx) and Huffman encoding algorithm reduces it to 75% (Approx).

<u>Table 1.4 -Time complexity</u>

File name	**File size (in bytes)**	**Huffman encoding algorithm**	**Shannon-fano algorithm**	**Lempel-ziv 77 algorithm**
input8.txt	448	0.000608	0.000824	0.001272
input10.txt	1384	0.000636	0.001064	0.001644
input1.txt	1936	0.001065	0.001093	0.002505
input2.txt	5264	0.000292	0.000370	0.001395
input4.txt	6856	0.002414	0.002788	0.012018
input6.txt	10456	0.003693	0.003008	0.018217
input3.txt	11776	0.004316	0.003003	0.024441
input5.txt	19392	0.002497	0.001584	0.011361
input7.txt	29864	0.002680	0.006254	0.020151
input9.txt	68152	0.004373	0.002717	0.042222

Conclusion for table 1.4

- As we increase file size the Time complexity of Huffman Algorithm increases linearly.
- As we increase file size the Time complexity of Shannonfano Algorithm increases or decreases depending on type of data.
- As we increase file size the Time complexity of LZ77 Algorithm increases abruptly.

Table 1.5 -Compression ratio

File name	File size (in bytes)	Huffman encoding algorithm	Shannon fano algorithm	Lempel-Ziv 77 algorithm
input8.txt	448	0.254464	0.392857	1.589286
input10.txt	1384	0.251445	0.393064	1.208092
input1.txt	1936	0.251033	0.421488	0.252066
input2.txt	5264	0.242021	0.484992	0.278116
input4.txt	6856	0.248541	0.441219	0.072345
input6.txt	10456	0.249809	0.436400	0.005356
input3.txt	11776	0.247113	0.467221	0.151495
input5.txt	19392	0.250103	0.467512	0.030116
input7.txt	29864	0.249933	0.468624	0.013126
input9.txt	68152	0.249677	0.486031	0.000704

Conclusion for table 1.5

- As we increase file size the Compression ratio of Huffman encoding Algorithm remains almost Constant .Average Compression ratio of Huffman encoding algorithm is 0.249414.
- As we increase file size the Compression ratio of Shannonfano Algorithm remains almost Constant .Average Compression ratio of Shannonfano algorithm is 0.3991896
- As we increase file size the Compression ratio of LZ77 Algorithm decreases abruptly.

Table 1.6-Compression factor

File name	File size (in bytes)	Huffman encoding algorithm	Shannon fano algorithm	Lempel-Ziv 77 algorithm
input8.txt	448	3.929825	2.545455	0.629214
input1.txt	1936	3.983539	2.372549	3.967213
input10.txt	1384	3.977011	2.544118	0.827751
input2.txt	5264	4.131866	2.061888	3.595628
input4.txt	6856	4.023474	2.266446	13.822580
input6.txt	10456	4.003063	2.291475	186.714279
input3.txt	11776	4.046735	2.140313	6.600897
input5.txt	19392	3.998351	2.138981	33.205479
input7.txt	29864	4.001072	2.133905	76.183670
input9.txt	68152	4.005172	2.057481	1419.83374

Conclusion for table 1.6

- As we increase file size the Compression factor of Huffman encoding Algorithm remains almost Constant .Average Compression factor of Huffman encoding algorithm is 4.0100108.
- As we increase file size the Compression factor of Shannonfano Algorithm remains almost Constant .Average Compression factor of Shannonfano algorithm is 2.2552611
- As we increase file size the Compression factor of LZ77 Algorithm increases abruptly.

Table 1.7-Encoded file size

File name	File size (in bytes)	Huffman encoding algorithm	Shannon fano algorithm	Lempel-ziv 77 algorithm
input8.txt	448	114	176	712
input10.txt	1384	348	544	1672
input1.txt	1936	486	816	488
input2.txt	5264	1274	2553	1464
input4.txt	6856	1704	3025	496
input6.txt	10456	2612	4563	56
input3.txt	11776	2910	5502	1784
input5.txt	19392	4050	9066	584
input7.txt	29864	7464	13995	392
input9.txt	68152	17016	33124	48

Conclusion for table 1.7

- For Huffman encoding algorithm the file size is always reduced by almost 74%-75%.
- For Shannonfano algorithm the file size is always reduced by around 55%-60%.
- As We Mentioned that Lempel-ziv 77 algorithmis Dictionary based Algorithm, the reduction of file size is highly dependent on data, if a word repeats several times in our Data File it reduces our file Size to 95%-98**%** ,otherwise for a very small file it sometimes even increases the file size.

Table 1.8-Saving percentage

File name	**File size (in bytes)**	**Huffman encoding algorithm**	**Shannon fano algorithm**	**Lempel-ziv 77 algorithm**
input8.txt	448	74.553574	60.714287	-58.928574
input10.txt	1384	74.855492	60.693638	-20.89248
input1.txt	1936	74.896690	57.851238	74.793388
input2.txt	5264	75.797874	51.500763	72.188446
input4.txt	6856	75.145859	55.878059	92.765465
input6.txt	10456	75.019127	56.359982	99.464424
input3.txt	11776	75.288727	53.277855	84.850540
input5.txt	19392	74.989685	53.248764	96.988449
input7.txt	29864	75.006699	53.137558	98.687378
input9.txt	68152	75.032280	51.396877	99.929573

Conclusion for table 1.8

- For Huffman encoding algorithm the saving percentage is almost 74%-75%.
- For Shannonfano algorithm the saving percentage is almost 55%-60%.
- As We Mentioned that Lempel-ziv 77 algorithm is Dictionary based Algorithm, saving percentage is highly dependent on data, if a word repeats several times in our Data File saving percentage increases to 95%-98**%** ,but for a very small file size sometimes saving percentage may even go negative.

Chapter 11
CONCLUSION

The end of this project brings us to the following Conclusions :

1) Since the exponential growth in data and limited storage space,data compression plays a major role in today’s world.

2) Any algorithm discussed above can't be termed as best or worst as they all depend on the type of data which has been compressed by them.

3) Each algorithm has its own advantages and disadvantages and has it's own applications.

4) The three algorithm discussed and implemented by us are conventional compression algorithms and their different subsidiaries with different optimizations have already been developed so there lies a limited scope in their optimization.

Chapter 12
REFERENCES

[1] Monika Soni , Dr Neeraj Shukla “Data Compression Techniques in Cloud Computing”

[2] Mohammad Hosseini “A Survey of Data Compression Algorithms and their Applications”

[3] Pu, I.M., 2006, Fundamental Data Compression, Elsevier, Britain.

[4] Kesheng, W., J. Otoo and S. Arie, 2006. Optimizing bitmap indices with efficient compression, ACM Trans. Database Systems, 31: 1-38.

[5] https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=15&ved=2ahUKEwjW8piM_q_dAhVHyrwKHXE2DHoQFjAOegQIABAC&url=http%3A%2F%2Fecehithaldia.in%2Fteaching_material%2FShanon-Fano1586521731.pdf&usg=AOvVaw0MHM4foSS-sDhzqyRAVfaE

[6] S.R. Kodituwakku ,U. S. Amarasinghe “Comparision of Lossless data compression algorithms for text data”

[7] Highly Efficient Compression Algorithms for Multichannel EEG,Laxmi Shaw , Student Member, IEEE , Daleef Rahman, and Aurobinda Routray, Senior Member, IEEE

[8] A Survey of Data Compression Algorithms and their Applications ,Mohammad Hosseini

[9] Network Systems Lab, School of Computing Science, Simon Fraser University, BC, Canada,Email: mohammad hosseini@sfu.ca

A . APPENDIX I PROJECT CODE

Huffman.c

```
#include <stdio.h>
#include <stdlib.h>
#include<time.h>
#define HEIGHT 100
int frequency[128]={0};
int temp[128];
int* ard[107];
int len;
typedef struct NODE Node;
struct NODE {
		char data;
		int freq;
		Node *left, *right;
};
typedef struct  {
		int size;
		int capacity;
		Node** array;
}MinHeap;

Node* newNode(char data, int freq)
{
		Node* temp= (Node*)malloc
(sizeof(Node));
		temp->left = temp->right = NULL;
		temp->data = data;
		temp->freq = freq;

		return temp;
}
void sort_huffman(int *frequency ,int *temp)
{
  int a,i,j;
  for (i = 0; i < 128; ++i) {
		if(!(frequency[i]>0.0))
		{
		continue;
		}
  for (j = i + 1; j < 128; ++j)
		{
		if (frequency[i] < frequency[j]) {
		a =  frequency[i];
		frequency[i] = frequency[j];
		frequency[j] = a;
		a=temp[i];
		temp[i]=temp[j];
```

```
            temp[j]=a;
  } } } }
void file_read(char * filename)
{
            FILE *fp;
            int length=0;
            int i;
            fp = fopen( filename, "r" ) ;
            if ( fp == NULL ){
            printf( "Could not open file %s\n",filename ) ; }
            int c;
            while((c = fgetc(fp))!=EOF){
            frequency[c]++;
            length++;
            }
            for(i=0;i<128;i++)
            {
            temp[i]=i; }
            sort_huffman(frequency,temp);
            fclose(fp);
}
MinHeap* createMinHeap(int capacity)
{
MinHeap* minHeap = (MinHeap*)malloc(sizeof(MinHeap));
            minHeap->size = 0;
            minHeap->capacity = capacity;
            minHeap->array
            = (Node**)malloc(minHeap->
capacity * sizeof(Node*));
            return minHeap;
}
void swapNode(Node** a,Node** b)
{
Node* t = *a;
*a = *b;
*b = t;
}
void minHeapify(MinHeap* minHeap, int idx)
{
            int smallest = idx;
            int left = 2 * idx + 1;
            int right = 2 * idx + 2;

            if (left < minHeap->size && minHeap->array[left]->
freq < minHeap->array[smallest]->freq)
            smallest = left;

            if (right < minHeap->size && minHeap->array[right]->
freq < minHeap->array[smallest]->freq)
            smallest = right;
            if (smallest != idx) {
            swapNode(&minHeap->array[smallest],
                        &minHeap->array[idx]);
            minHeapify(minHeap, smallest);
            } }
int isSizeOne(MinHeap* minHeap)
{

            return (minHeap->size == 1);
}

Node* extractMin(MinHeap* minHeap)
{
Node *temp = minHeap->array[0];
            minHeap->array[0]= minHeap->array[minHeap->size - 1];

            --minHeap->size;
            minHeapify(minHeap, 0);
           return temp;
}

void insertMinHeap(MinHeap* minHeap,Node* minHeapNode)
{
            ++minHeap->size;
            int i = minHeap->size - 1;
             while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {
              minHeap->array[i] = minHeap->array[(i - 1) / 2];
            i = (i - 1) / 2;
            }
            minHeap->array[i] = minHeapNode;
}
void buildMinHeap(MinHeap* minHeap)
{

            int n = minHeap->size - 1;
            int i;

            for (i = (n - 1) / 2; i >= 0; --i)
            minHeapify(minHeap, i);
}

void printArr(int arr[], int n)
{
            int i;
            ard[len]=(int*)malloc(sizeof(int)*n);
            for(i=0;i<n;i++)
            ard[len][i]=arr[i];
            len++;
}
int isLeaf(Node* root)
{
```

```
            return !(root->left) && !(root->right);
}
MinHeap* createAndBuildMinHeap(char data[], int freq[], int size)
{
   int i;
   MinHeap* minHeap = createMinHeap(size);

            for ( i = 0; i < size; ++i)
            minHeap->array[i] = newNode(data[i], freq[i]);

            minHeap->size = size;
            buildMinHeap(minHeap);
            return minHeap;
}
Node* buildHuffmanTree(char data[], int freq[], int size)
{
            Node *left, *right, *top;
            MinHeap* minHeap = createAndBuildMinHeap(data, freq, size);

            while (!isSizeOne(minHeap)) {
            left = extractMin(minHeap);
            right = extractMin(minHeap);
            top = newNode('$', left->freq + right->freq);
            top->left = left;
            top->right = right;

            insertMinHeap(minHeap, top);
            }
            return extractMin(minHeap);
}
void printCodes(Node* root, int arr[], int top)
{
int i=0;
int j=0;
            if (root->left) {

            arr[top] = 0;
            printCodes(root->left, arr, top + 1);
            }

            if (root->right) {

            arr[top] = 1;
            printCodes(root->right, arr, top + 1);
            }
            if (isLeaf(root)) {
            printArr(arr, top);
            }
}
void HuffmanCodes(char data[], int freq[], int size)
{
```

```
            Node* root
            = buildHuffmanTree(data, freq, size);
            int arr[HEIGHT], top = 0;

            printCodes(root, arr, top);
}
double executeHuffman(char * filename)
{
  clock_t start,end;
double total_time;
start=clock();
len=0;
char c;
int i=0 ,j=0,k=0;
char arr[128];
int freq[128];
file_read(filename);
for(i=0;i<128;i++)
{
if(frequency[i]>0)
{
freq[k]=frequency[i];
arr[k]=(char)temp[i];
k++;
}
}
int size =k;
HuffmanCodes(arr, freq, size);
FILE *fp;
 FILE *fptr;
            if ((fptr = fopen("outputHuffman.txt","w")) == NULL){
            printf("Error! opening file");
            // Program exits if the file pointer returns NULL.
            return 0;
            }
            fp = fopen( filename, "r" ) ;
            if ( fp == NULL ){
            printf( "Could not open file %s\n" ,filename) ;
            return 1;
            }
            while((c = fgetc(fp))!=EOF){
            for ( i = 0; i <size; ++i)
            {
            if(arr[i]==c){
            for(j=0;j<sizeof(ard[i])/sizeof(int);j++)
            {
            fprintf(fptr,"%d",ard[i][j]);
            }}}}
fclose(fp);
fclose(fptr);
end=clock();
```

```
total_time=(double)(end-start)/CLOCKS_PER_SEC;
return total_time;
}
```

Shannonfano.c

```
#include<string.h>
#include<stdio.h>
#include<time.h>
int codeword[128]={0};
void sort(int *frequency ,int *temp)
{
  int a,i,j;
  for (i = 0; i < 128; ++i) {
            for (j = i + 1; j < 128; ++j)
            {
            if (frequency[i] < frequency[j])
            {
            a =  frequency[i];
            frequency[i] = frequency[j];
            frequency[j] = a;
            a=temp[i];
            temp[i]=temp[j];
            temp[j]=a;
            }
            }
  }
}
void shannonfano(int initial,int final,int code,float *probability,int *character)
{
  if(initial==final)
  {

            codeword[initial]=code;
            return ;
  }
  int first=initial;
  int last=final;
  float sumFront=probability[first];
  float sumLast=probability[last];
  while(first!=last)
  {
            if(sumFront>=sumLast)
            {
            last--;
            sumLast=sumLast+probability[last];
            }
            else{
            first++;
            sumFront=sumFront+probability[first];
            }
  }
  code=code*10;
  shannonfano(initial,first,code+0,probability,character);
  shannonfano(first+1,final,code+1,probability,character);
}
double executeShanonfano(char * filename){
  clock_t start,end;
  double total_time;
  start=clock();
            FILE *fp;
            int length=0;
            int frequency[128]={0};
            int i;
            int temp[128];
            fp = fopen( filename, "r" ) ;
            if ( fp == NULL ){
            printf( "Could not open file %s\n",filename ) ;
            return 1;
            }
            int c;
            while((c = fgetc(fp))!=EOF){
            frequency[c]++;
            length++;
            }
            for(i=0;i<128;i++)
            {
            temp[i]=i;
            }
            sort(frequency,temp);
            float probab[128]={0.0};
            int last=0;
            for(i=0;i<128;i++)
            {
            if(frequency[i]==0)
            {
            last=i;
            break;
            }
            probab[i]=frequency[i]/(float)length;
            }
            shannonfano(0,last-1,0,probab,temp);
            fclose(fp);
            //starting to write in binary file
            FILE *fptr;
            if ((fptr = fopen("outputShanonfano.txt","w")) == NULL){
            printf("Error! opening file");
            // Program exits if the file pointer returns NULL.
            return 0;
            }
            fp = fopen( filename, "r" ) ;
```

```
        if ( fp == NULL ){
        printf( "Could not open file input.txt\n" ) ;
        return 1;
        }
        while((c = fgetc(fp))!=EOF){
        for (int i = 0; i <last; ++i)
        {
        if(temp[i]==c)
        {
        fprintf(fptr,"%d",codeword[i]);
        }}}
        fclose(fp);
        fclose(fptr);
        end=clock();
        total_time=(double)(end-start)/CLOCKS_PER_SEC;
        return total_time;
}
```

LZ.c

```
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <math.h>
uint32_t lz77_compress (uint8_t *uncompressed_text,uint32_t uncompressed_size,uint8_t
*compressed_text, uint8_t pointer_length_width)
{
        uint16_t pointer_pos, temp_pointer_pos, output_pointer, pointer_length,
temp_pointer_length;
        uint32_t compressed_pointer, output_size, coding_pos, output_lookahead_ref,
look_behind, look_ahead;
        uint16_t pointer_pos_max, pointer_length_max;
        pointer_pos_max = pow(2, 16 - pointer_length_width);
        pointer_length_max = pow(2, pointer_length_width);

        *((uint32_t *) compressed_text) = uncompressed_size;
        *(compressed_text + 4) = pointer_length_width;
        compressed_pointer = output_size = 5;

        for(coding_pos = 0; coding_pos < uncompressed_size; ++coding_pos)
        {
        pointer_pos = 0;
        pointer_length = 0;
        for(temp_pointer_pos = 1; (temp_pointer_pos < pointer_pos_max) &&
(temp_pointer_pos <= coding_pos); ++temp_pointer_pos)
        {
        look_behind = coding_pos - temp_pointer_pos;
        look_ahead = coding_pos;
        for(temp_pointer_length = 0; uncompressed_text[look_ahead++] ==
uncompressed_text[look_behind++]; ++temp_pointer_length)
```

```
			if(temp_pointer_length == pointer_length_max)
			break;
		if(temp_pointer_length > pointer_length)
		{
			pointer_pos = temp_pointer_pos;
			pointer_length = temp_pointer_length;
			if(pointer_length == pointer_length_max)
			break;
		}
		}
		coding_pos += pointer_length;
		if((coding_pos == uncompressed_size) && pointer_length)
		{
		output_pointer = (pointer_length == 1) ? 0 : ((pointer_pos << pointer_length_width) |
(pointer_length - 2));
		output_lookahead_ref = coding_pos - 1;
		}
		else
		{
		output_pointer = (pointer_pos << pointer_length_width) | (pointer_length ?
(pointer_length - 1) : 0);
		output_lookahead_ref = coding_pos;
		}
		*((uint16_t *) (compressed_text + compressed_pointer)) = output_pointer;
		compressed_pointer += 2;
		*(compressed_text + compressed_pointer++) = *(uncompressed_text +
output_lookahead_ref);
		output_size += 3;
		}
return output_size;
}
uint32_t lz77_decompress (uint8_t *compressed_text, uint8_t *uncompressed_text)
{
		uint8_t pointer_length_width;
		uint16_t input_pointer, pointer_length, pointer_pos, pointer_length_mask;
		uint32_t compressed_pointer, coding_pos, pointer_offset, uncompressed_size;
		uncompressed_size = *((uint32_t *) compressed_text);
		pointer_length_width = *(compressed_text + 4);
		compressed_pointer = 5;

		pointer_length_mask = pow(2, pointer_length_width) - 1;

		for(coding_pos = 0; coding_pos < uncompressed_size; ++coding_pos)
		{
		input_pointer = *((uint16_t *) (compressed_text + compressed_pointer));
		compressed_pointer += 2;
		pointer_pos = input_pointer >> pointer_length_width;
		pointer_length = pointer_pos ? ((input_pointer & pointer_length_mask) + 1) : 0;
		if(pointer_pos)
		for(pointer_offset = coding_pos - pointer_pos; pointer_length > 0; --pointer_length)
			uncompressed_text[coding_pos++] = uncompressed_text[pointer_offset++];
		*(uncompressed_text + coding_pos) = *(compressed_text + compressed_pointer++);
		}
return coding_pos;
}
long fsize (FILE *in)
{
	long pos, length;
	pos = ftell(in);
	fseek(in, 0L, SEEK_END);
	length = ftell(in);
	fseek(in, pos, SEEK_SET);
	return length;
}
uint32_t file_lz77_compress (char *filename_in, char *filename_out, size_t malloc_size,
uint8_t pointer_length_width)
{
	FILE *in, *out;
	uint8_t *uncompressed_text, *compressed_text;
	uint32_t uncompressed_size, compressed_size;

	in = fopen(filename_in, "r");
	if(in == NULL)
	return 0;
	uncompressed_size = fsize(in);
	uncompressed_text = malloc(uncompressed_size);
	if((uncompressed_size != fread(uncompressed_text, 1, uncompressed_size, in)))
	return 0;
	fclose(in);
	compressed_text = malloc(malloc_size);
	compressed_size = lz77_compress(uncompressed_text, uncompressed_size,
compressed_text, pointer_length_width);

	out = fopen(filename_out, "w");
	if(out == NULL)
	return 0;
	if((compressed_size != fwrite(compressed_text, 1, compressed_size, out)))
	return 0;
	fclose(out);
	return compressed_size;
}
double total_time=0.0;
double timecomplexity()
{
  return total_time;

}
double executeLZ (char * filename)
{clock_t start,end;
//double total_time;
start=clock();
	FILE *in;
```

```
        in = fopen(filename, "r");
        if(in == NULL)
        return 0;
        fclose(in);
        uint8_t arr[4];
        for(uint8_t i = 1; i <4 ; ++i)
        {
        arr[i]=file_lz77_compress(filename, "outputLZ.txt", 10000000, i);
        }
        end=clock();
        total_time=(double)(end-start)/CLOCKS_PER_SEC;
        return arr[3]*8;
}
```

Main.c

```
#include "shanonfano.c"
#include "huffman.c"
#include "lz.c"
int main()
{
        char c;
        printf("Enter Input File Name with extension\n");
        char filename[90];
        scanf("%s",filename );
        double
compressionRatioHuffman=0.0,compressionRatioShanonfano=0.0,compressionRatioLz=0.0;
        double
compressionFactorHuffman=0.0,compressionFactorShanonfano=0.0,compressionFactorLz=0.0
;
        double
savingPercentageHuffman=0.0,savingPercentageShanonfano=0.0,savingPercentageLz=0.0;
        int lengthShannonfano=0,lengthHuffman=0,lengthInputFile=0;
        int
sizeOfInputFile=0 ,sizeOfShanonfanoEcodedFile=0,sizeOfHuffmanEcodedFile=0,sizeOfLzEc
odedFile=0;
        double timeTakenShanonfano=executeShanonfano(filename);
        double timeTakenHuffman =executeHuffman(filename);
        sizeOfLzEcodedFile=executeLZ(filename);
        double timeTakenLz=timecomplexity();
        FILE *inputFile = fopen(filename, "r" ) ;
        if ( inputFile== NULL ){
        printf( "Could not open file input.txt\n" ) ;
        return 1;
        }
        while((c = fgetc(inputFile))!=EOF){
        lengthInputFile++;
        }
        sizeOfInputFile=(lengthInputFile-1)*8;
        FILE *outputShanonfano = fopen( "outputShanonfano.txt", "r" ) ;
        if ( outputShanonfano== NULL ){
        printf( "Could not open file input.txt\n" ) ;
        return 1;
        }
        while((c = fgetc(outputShanonfano))!=EOF){
        lengthShannonfano++;
        }
        sizeOfShanonfanoEcodedFile=lengthShannonfano*1;
        FILE *outputHuffman = fopen( "outputHuffman.txt", "r" ) ;
        if ( outputHuffman== NULL ){
        printf( "Could not open file input.txt\n" ) ;
        return 1;
        }
        while((c = fgetc(outputHuffman))!=EOF){
        lengthHuffman++;
        }
        sizeOfHuffmanEcodedFile=lengthHuffman*1;
        compressionRatioHuffman=sizeOfHuffmanEcodedFile/(float)sizeOfInputFile;
        compressionRatioShanonfano=sizeOfShanonfanoEcodedFile/(float)sizeOfInputFile;
        compressionRatioLz=sizeOfLzEcodedFile/(float)sizeOfInputFile;
        compressionFactorHuffman=sizeOfInputFile/(float)sizeOfHuffmanEcodedFile;
        compressionFactorShanonfano=sizeOfInputFile/(float)sizeOfShanonfanoEcodedFile;
        compressionFactorLz=sizeOfInputFile/(float)sizeOfLzEcodedFile;
        savingPercentageHuffman=((sizeOfInputFile-sizeOfHuffmanEcodedFile)/
(float)sizeOfInputFile)*100;
        savingPercentageShanonfano=((sizeOfInputFile-sizeOfShanonfanoEcodedFile)/
(float)sizeOfInputFile)*100;
savingPercentageLz=((sizeOfInputFile-sizeOfLzEcodedFile)/(float)sizeOfInputFile)*100;
        printf("time complexity huffman %lf\n",timeTakenHuffman );
        printf("time complexity shanonfano %lf\n",timeTakenShanonfano );
        printf("time complexity LZ %lf\n",timeTakenLz );
        printf("sizeOfInputFile %d\n",sizeOfInputFile );
        printf("sizeOfHuffmanEcodedFile %d\n",sizeOfHuffmanEcodedFile );
        printf("sizeOfShanonfanoEcodedFile %d\n",sizeOfShanonfanoEcodedFile );
        printf("sizeOfLzEcodedFile %d\n",sizeOfLzEcodedFile );
        printf("compressionRatioHuffman %lf\n",compressionRatioHuffman );
        printf("compressionRatioShanonfano %lf\n",compressionRatioShanonfano );
        printf("compressionRatioLz %lf\n",compressionRatioLz );
        printf("compressionFactorHuffman %lf\n",compressionFactorHuffman );
        printf("compressionFactorShanonfano %lf\n",compressionFactorShanonfano );
        printf("compressionFactorLz %lf\n",compressionFactorLz );
        printf("savingPercentageHuffman %lf\n",savingPercentageHuffman );
        printf("savingPercentageShanonfano %lf\n",savingPercentageShanonfano );
        printf("savingPercentageLz %lf\n",savingPercentageLz);}
```

Report Verified by

Mr. G.L. Prakash	**Mr. Amit Agarwal**
Project Guide	**HOD**
(Name & Sign)	**(Dept. of Virtualization)**