Implementation of Various Lossless Data Compression Algorithms And Comparison And Analysis Of Their Results.

Project report submitted in partial fulfillment of the requirement for the degree of Bachelor of Technology

In

Computer Science and Technology

and

Information Technology

By

Puru Aggarwal 141248 Tista Lather 141402

Under the supervision of

Prof. Dr. Satya Prakash Ghrera,
FBCS, SMIEEE
Professor, Brig (Retd.) and Head, Dept. of CSE and IT

Department of Computer Science & Engineering and Information Technology

Jaypee University of Information Technology Waknaghat, Solan-173234, Himachal Pradesh

Certificate

Candidate's Declaration

I hereby declare that the work presented in this report entitled **"Implementation of Lossless Data Compression Algorithms"** in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Information Technology** submitted in the Department of Computer Science & Engineering and Information Technology**,** Jaypee University of Information Technology Waknaghat is an authentic record of my own work carried out over a period from August 2017 to December 2017 under the supervision of **Brig.(Retd) Prof. Dr. Satya Prakash Ghrera,Professor and Head, Dept. of CSE and IT.**

The work done embodied in the report has not been appeased for the award of any other degree or diploma.

(Student Signature) (Student Signature)

Puru Aggarwal 141248 Tista Lather 141402

This is to certify that the above affirmation made by the candidate is true to the best of my knowledge.

(Supervisor Signature)

Brig.(Retd) Prof. Dr. Satya Prakash Ghrera

Professor, Department of CSE & IT

Dated:

ACKNOWLEDGEMENTS

In implementing our project, we had to take the help and guideline of some respected persons, who deserve our greatest gratitude. We would like to show our gratitude to Prof.Dr.Satya Prakash Ghrera, FBCS,SMIEEE professor , Brig(Retd.) and Head of Dept. of CSE and IT for giving us a good guideline for project throughout numerous consultations. We would also like to expand our deepest gratitude to all those who have directly and indirectly guided us in our project.

Puru Aggarwal	Tista Lather
141248	141402

Contents Index

LIST OF FIGURES& TABLES

CHAPTER-I
INTRODUCTION

1.1 INTRODUCTION

Data compression refers to a field in information theory and coding which primarily deals with minimization of the amount of data for information transmission and storage. The basic character of data compression techniques remains to represent given set of data in the form of characters into another form with data length comparatively of smaller size. The objective is achieved via removal of redundancies in data representation in order to decrease the information transmission and storage requirements. This reduction of redundancies refers to data compression. The major benefit of using data compression is in reduced information storage space required as well as reduced transmission channel capacity.

Data compression is majorly categorized into two: Lossless and Lossy compression. Lossless compression is primary used for text compression as textual information requires exact reproduction of the original text sequence. It means that the compressed message has the same quantity of information but represented or transmitted using fewer characters. Whereas , lossless data compression is used for audio, video and image signals. The requirement is imperative as a bit compromise on quality of such signals may be tolerated over space and channel capacity requirements.

In this project work, different lossless data compression techniques have been studied and are compared using a single standard text data set. The comparisons have been listed and conclusion for further project work have been drawn.

1.2 DATA COMPRESSION TECHNIQUES – FUNDAMENTAL CONCEPTS

In this section, different concepts involved in data compression have been studied and discussed briefly. The terms, concepts and definitions discussed are fundamental to the study of data compression algorithms.

1.2.1 Information Theory and Coding

Information theory is a one of branches of applied mathematics that deals with complexity and interference in transmission of data. It measures amount of information in data and how easily it can be adjudged by someone who is unaware of its value. The key measures of information theory are entropy, mutual information and channel capacity etc.

Entropy

Entropy is the key measure in information theory and represents the amounts of uncertainty involved in outcome of a random experiment. Let S is the set of all possible messages coming from a source, and p_i be the probability of the $i-th$ message. Then, the Shannon entropy denoted as H (bits per symbol) was defined based on probability density function of each communicated source symbol as

$$H = -\sum_i p_i \log_2(p_i)$$

Where p_i refers to probability of the occurrence of the $i-th$ possible outcome of the symbol source.

1.3 CLASSIFICATION OF DATA COMPRESSION TECHNIQUES

The categorization of data compression techniques is as shown in the Figure 1.1.

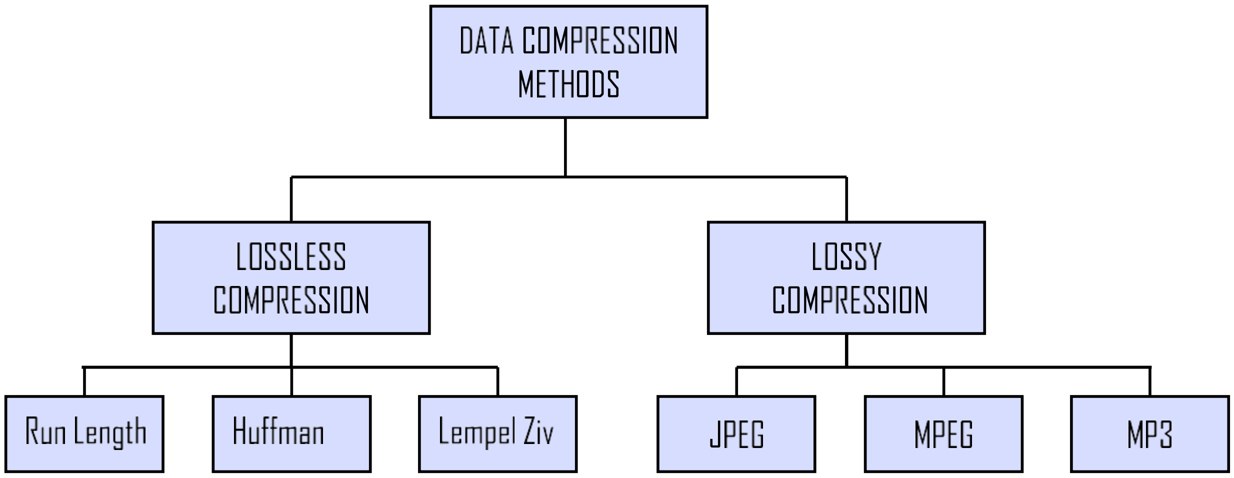


Figure 1.1: Classification of Data Compression Methods: Lossless and Lossy

The Data compression techniques are primarily categorized into two: Lossless and Lossy compression techniques. Certain lossless data compression algorithms are based upon dictionary or adaptive dictionary. Generally, a sequence of previously coded sequence is treated as dictionary. Lempel and Ziv proposed two algorithms LZ77 and LZ78 to implement the dictionary methods.

Huffman algorithm inputs a list of non-negative weights representative of the frequency or cost of the message symbols. This coding construct a full binary labelled tree using weights. The formation of tree starts with singleton tree constructed using each weight in the list.Run-length encoding (RLE) is a very simple form of lossless data compression in which runs of data are stored as a single data value and count, rather than as the original run.

1.4 PROJECT OBJECTIVES

The project work aims to study and implement some lossless data compression algorithms. Following algorithms have been comprehensively studied, implemented and compared.

- Shannon Fano Coding
- LZ77 Algorithm
- Deflate Algorithm
- Huffman Coding
- Run Length Coding
- LZW Algorithm
- Burrows-Wheeler Algorithm
- Arithmetic Encoding Algorithm

The objective of the present work remains to implement lossless data compression algorithms on standard data sets. The study will involve implementation of the algorithm using Java and C++. Further, the results from various compression algorithms will be compared with results reported.

1.5 PROJECT METHODOLOGY

To achieve the objectives of the project work, the following methodology has been adopted

- Study of the fundamental concepts in Information Theory and Coding
- Study of Literature available in the area
- Coding of the Lossless Compression Algorithms
- Continued Literature Survey and Content Writing
- LZ77 Implementation
- Huffman Coding
- Run Length Coding
- LZW Algorithm
- Burrows-Wheeler Algorithm

- Arithmetic Encoding Algorithm
- Shannon Fano Encoding Algorithm
- Deflate Algorithm Implementation &Report Submission

The phase-wise methodology adopted has been shown in Figure 1.2 and 1.3

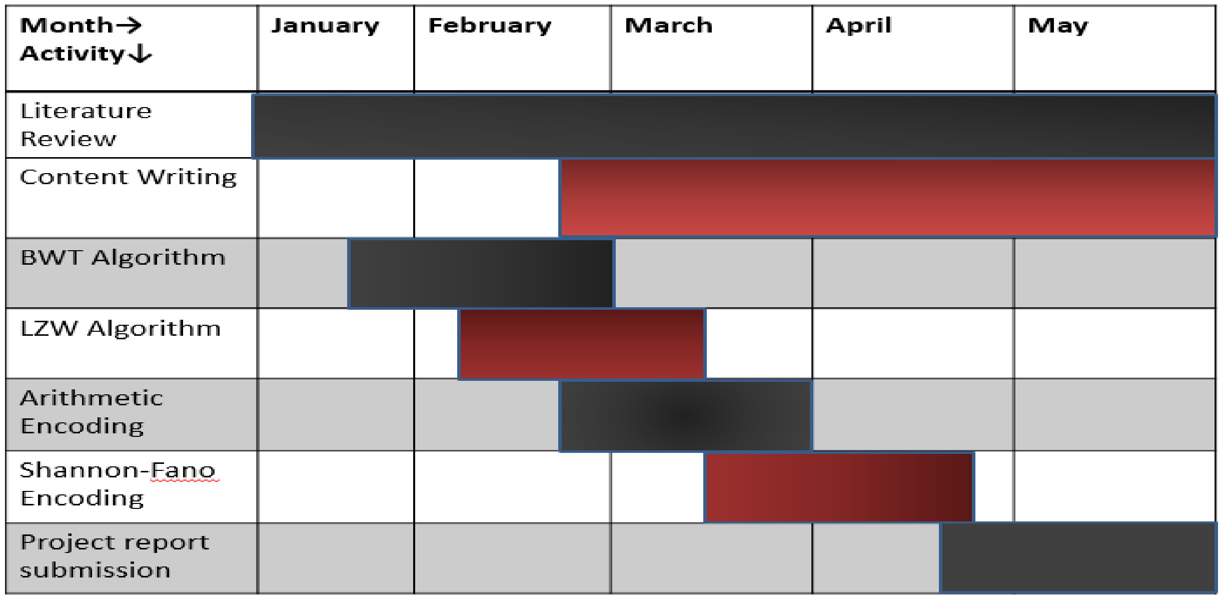


Fig- 1.2

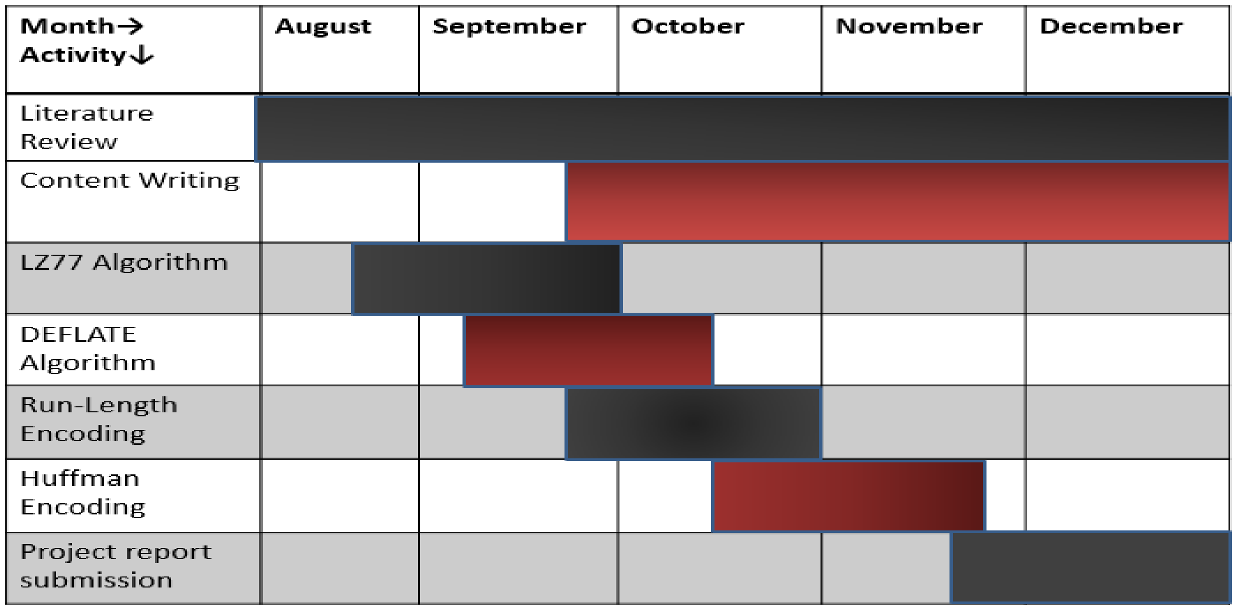


Fig- 1.3

1.6 ORGANIZATION OF PROJECT WORK

Chapter 2 of this project report deals with literature survey on project topic. Archival literature exists on the information theory and coding. Some of the earlier reference in this area were studied. The landmark algorithms LZ77, Huffman coding, Shannon-Fano coding and LZW were studied from the literature. Further, advanced algorithms like deflate and inflate and burrows wheeler algorithms were studied, and finally run length coding was studied. Few current research papers on application of data compression techniques for wireless sensor networks and biomedical image compression were also referred to.

Chapter 3 of this project report deals with detailed discussion on four lossless data compression algorithms studied and implemented. Various internet sites and literature was referred to for understanding these techniques. Initial studies were done on trivial strings and later the ideas were implemented using Java

Chapter 4 of this project report deals with the performance metrics of the data compression algorithms. Several performance measures for the data compression algorithms were discussed. Compression ratio was chosen as metric in this project for comparison of various data compression algorithms.Four lossless data compression program output was compared in terms of compression ratio with results presented in the form of a table.

Chapter 5 Discusses the conclusions on the basis of the study and results taken. The chapter is followed by the references as well as web references.

Towards the end of this report, the dataset taken up for the study as well as source code of all four lossless data compression algorithms are listed in the form of Appendix.

CHAPTER II

LITERATURE REVIEW

Data compression has remained celebrated topic of research in initial phase of information technology due to compressive need of time and space optimization. Redundancy in information was exploited using the principles of information theory and coding to cater the demand of data compression. The data compression was achieved by removal of redundancy in representation of data. Although, data compression algorithm types: Lossy and lossless, both are equally important, yet only lossless data compression were studied in this project. The complementary part shall be taken up as further study in this area. The considered lossless data compression algorithms for the project are:-

- LZ77 Algorithm
- Huffman Encoding Algorithm
- Deflate Algorithm
- Run Length Coding Algorithm
- LZW Algorithm
- Burrows-Wheeler Algorithm
- Arithmetic Coding Algorithm
- Shannon-Fano Encoding

The study starts with the information and coding theory by Shannon[8] wherein the concept of coding theory was provided. Shannon formally floated the notion of capacity of a channel and related theorem for information channel capacity. Several other concepts like discrete noisy channels were also introduced. This text is widely referred and studied among the researchers in information theory and allied areas.

The project study starts with comprehensive study of initial chapters of K. Sayood[1] and D. Solomon[2]. Several models like physical and probability models were studied for representation of data. Coding of information using different coding schemes was also studied followed by some useful insights into algorithmic information theory. Further, various types of Huffman codes were also studied. This was followed by the study of dictionary techniques for data compression.

Ziv and Lempel[5] in 1977 presented the LZ77 algorithm which is a dictionary based lossless data compression technique. This algorithm later formed the basis of several data compression techniques. This method is based upon the dictionary based coders and maintains a sliding window during compression. This algorithm achieves encoding and decoding using a sliding window over previously observed characters; therefore, the process of decompression must start from the beginning of the compressed file.

Burrows Wheeler[7] exhibited a method that accomplishes compression within a percent or so of that accomplished by factual modeling techniques, however at speeds similar to those of calculations in view of Lempel and Ziv's[5,6]. Their calculation does not process its info consecutively, but rather forms a square of content as a solitary unit. The thought is to apply a reversible change to a square of content to frame another piece that contains similar characters, yet is simpler to pack by basic pressure calculations. The change tends to amass characters together with the goal that the likelihood of finding a character near another example of a similar character is expanded considerably.The quasi method for coding was shown to perform fairly fast but near optimal.

Huffman [3] provided a method for lossless data compression that assisted variety of other data compression programs. This method is almost similar to Shannon Fano coding, and it results in the best code when symbol probabilities are negative powers of 2. This method constructs the code tree using bottom up approach as compared to top down approach taken by Shannon Fano coding.

The project topic remains related to basic data compression algorithm, their implementation and performance comparison. Some of the methods proposed in literature exist as Patents where some alternatives exist in literature.

Several Web resources were consulted for ready reference on various data compression techniques as well as their implementation guidelines. Such web references have been listed in the list of references towards the end of this report.

CHAPTER III

LOSSLESS DATA COMPRESSION ALGORITHMS

Lossless data compression algorithms have been taken up for project study. These algorithms are as:

LZ77 Algorithm

Huffman Coding Algorithm

Deflate Algorithm

Burrows-Wheeler Transform

LZW Algorithm

Shannon-Fano Coding Algorithm

Run Length Encoding

Arithmetic Coding Algorithm

In this chapter, the introduction and working details of these lossless data compression techniques are discussed using simple examples.

3.1 LZ77 ALGORITHM

LZ77 algorithm was given by Jacob Jiv and Abraham Lempel [5] in 1977, which later shaped the premise of a few information pressure strategies. This strategy depends on the word reference coders and keeps up a sliding window amid pressure. This calculation accomplishes encoding and deciphering utilizing a sliding window over beforehand watched characters; thusly, the procedure of decompression must begin from the earliest starting point of the packed record.

LZ77 algorithm does pressure by supplanting rehashed events of images or information with references to remarkable duplicate of that image in uncompressed information document. The separation is otherwise called the balanced. The length-remove combine infers that each of next length characters leveling with precisely counterbalance elements behind it in the uncompressed stream.

3.1.1 LZ77 Compression Process

The sliding window in LZ77 consists of two portions

- Look-ahead Buffer
- Search Buffer

The Search bufferis the portion of already encoded the input string or sequence in which search for the symbols or sequences is done. The search buffer is searched backwardly and last character is denoted as 1 and then preceding characters are numbered in ascending order.

The Look-Ahead Buffer refers to the portion of sequence to be encoded next. The searching in the search buffer is done considering the symbols or sequence of symbols in the Look-Ahead buffer.

LZ77 Encoding: The LZ77 encoder takes the first symbol in the Look-ahead buffer and then searches it in search buffer backwardly till the match is found. As the symbolsdiscussed in search buffer are numbered backwardly, number corresponding to first occurrence of symbol in search buffer is noted. Initially the algorithm starts with assuming length of the search buffer and the Look Ahead Buffer. The initial string of symbols equal to length of the search buffer is encoded as such.

- The next symbol in the Look-Ahead buffer is chosen as the search symbol and is searched backward in the search buffer.
- Once the match with the largest offset is obtained, the encoder searches the search buffer with the Look-Ahead buffer symbol by symbol. The length of the largest string of symbols in the search buffer that matches string of symbols in Look-Ahead Buffer is known as length. Then the symbol(s) is/are encoded as (Offset, Length, Code(next character))

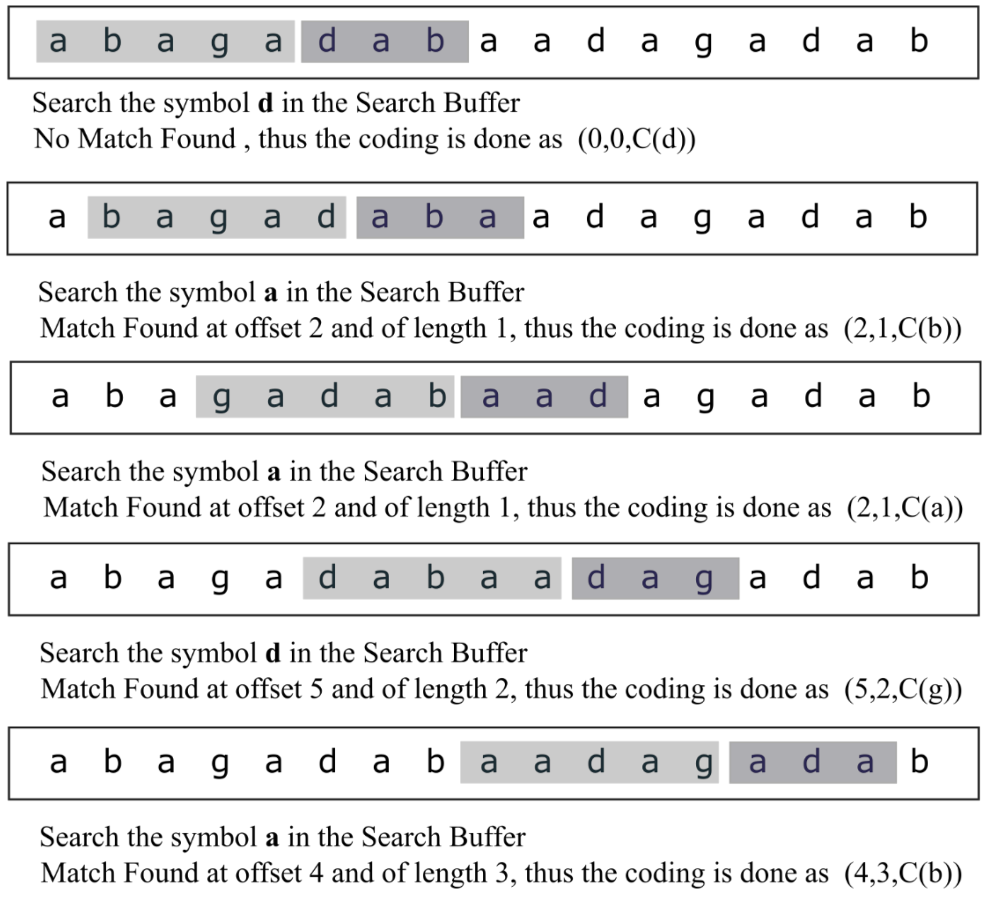
The Pseudocode for the LZ77 algorithm is given as

```
begin
fill the view from the input
    while (the view is filled) do
    begin
```

```
          find the larget prefix q of view starting in
thecoded part
a := position of q in window
b := length of q
c := first char after q in view
output(a,b,c)
add b+1 characters
     end
```

3.1.2 LZ77 Illustrative Example

The concept of the LZ77 algorithm as discussed has been illustrated using an example string `abagadabaadagadab` using Search buffer with length of 5 characters and Look-ahead buffer of 3 characters. The compression using LZ77 is illustrated in Figure 3.1.


a b a g a d a b a a d a g a d a b
a b a g a d a b a a d a g a d a b
a b a g a d a b a a d a g a d a b
a b a g a d a b a a d a g a d a b
a b a g a d a b a a d a g a d a b

Search the symbol **d** in the Search Buffer
No Match Found , thus the coding is done as (0,0,C(d))

Search the symbol **a** in the Search Buffer
Match Found at offset 2 and of length 1, thus the coding is done as (2,1,C(b))

Search the symbol **a** in the Search Buffer
Match Found at offset 2 and of length 1, thus the coding is done as (2,1,C(a))

Search the symbol **d** in the Search Buffer
Match Found at offset 5 and of length 2, thus the coding is done as (5,2,C(g))

Search the symbol **a** in the Search Buffer
Match Found at offset 4 and of length 3, thus the coding is done as (4,3,C(b))

Figure 3.1: LZ77 Compression Step for an example string abagadabaadagadab using Search buffer of 5 characters and Look-ahead buffer of 3 characters

The decompression step of the LZ77 algorithm proceeds as follows:

- The original string equal to the length of the search string is decoded as such in the initialization phase. abaga
- The first encoded information (0,0, C(d)) implies that no match was found during the compression step, and the string is appended with the character ‘d’. abagad
- Further, the encoded information (2,1,C(b)) implies that we move two offset back in the decoded string and then copy only one character followed by new character ‘b’. abagadab
- The same steps continue till we reach the end of the encoded information as shown in Figure 3.2.

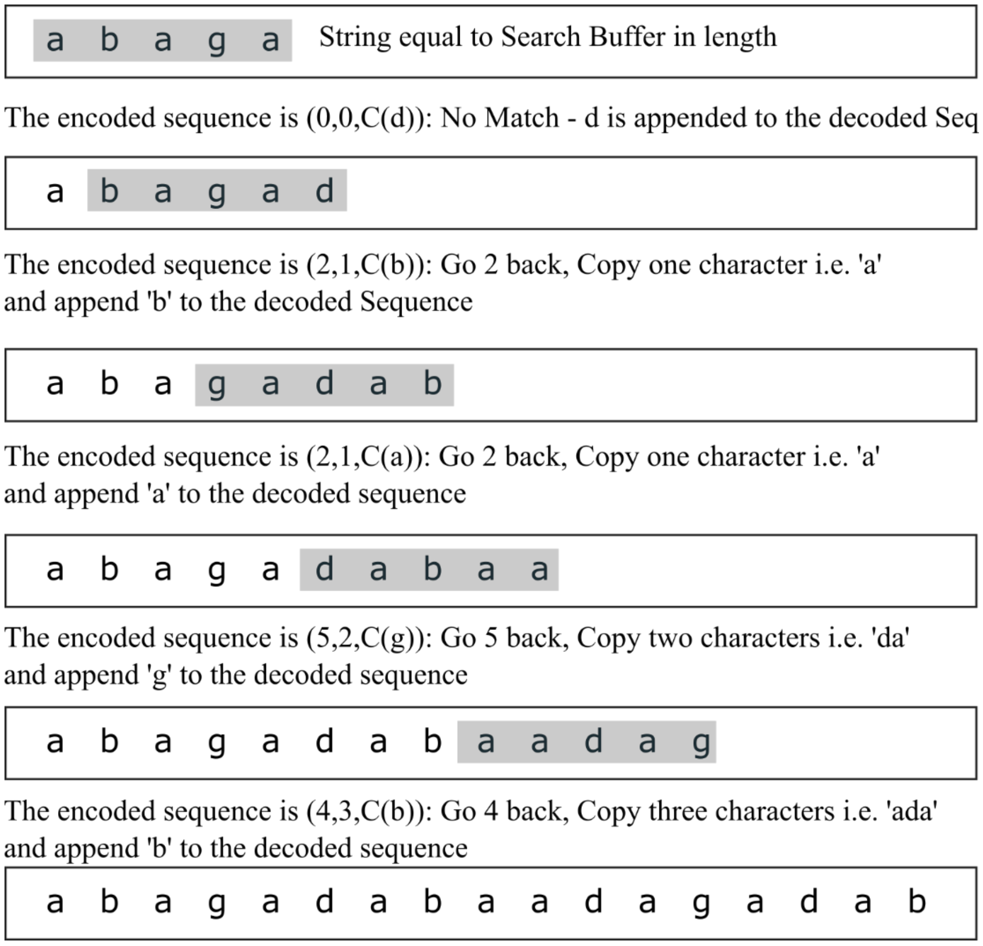

a b a g a
String equal to Search Buffer in length
The encoded sequence is (0,0,C(d)): No Match - d is appended to the decoded Seq
a b a g a d
The encoded sequence is (2,1,C(b)): Go 2 back, Copy one character i.e. 'a' and append 'b' to the decoded Sequence
a b a g a d a b
The encoded sequence is (2,1,C(a)): Go 2 back, Copy one character i.e. 'a' and append 'a' to the decoded sequence
a b a g a d a b a a
The encoded sequence is (5,2,C(g)): Go 5 back, Copy two characters i.e. 'da' and append 'g' to the decoded sequence
a b a g a d a b a a d a g
The encoded sequence is (4,3,C(b)): Go 4 back, Copy three characters i.e. 'ada' and append 'b' to the decoded sequence
a b a g a d a b a a d a g a d a b

Figure 3.2 LZ77 Decompression Step for an example string abagadabaadagadab using Search buffer of 5 characters and Look-ahead buffer of 3 characters

3.1.3 Advantages of LZ77 Algorithm

- The LZ77 algorithm offers the following advantages over others
- LZ77 is a simple form of adaptive compression algorithm that requires nil apriori information about the source or the characteristics of the source.
- Theoretically Limpel and Ziv proposed that rendering of the algorithm asymptotically approaches the best
- The LZ class of compression algorithm offer tremendous decompression speed. This property makes these algorithms particularly suited to compressed text in databases.

3.1.4 Disadvantages of LZ77 Algorithm

- The LZ77 suffers from the following disadvantages
- The algorithm works on the principle that patterns recur frequently or they occur/repeat too closely. If this is not the case then performance of the algorithm suffers.
- LZ77 has restricted size look-ahead buffer due to improve the speed of the encoder. If longer size of look-ahead buffer is chosen, the speed of encoder will be too slow.

3.2 **HUFFMAN ENCODING**

The method that is used for Huffman coding results in a prefix-free program. A prefix-free program is one in which the bit coding sequence represents some character which is not a prefix of the bit coding sequence representing some other character. For instance , A bit sequence for a Huffman code on an alphabet with 4 characters where Dis the most possible and A is the least possible:

A	110
D	0
C	10

B 111

3.2.1 Working of Huffman Coding

Huffman coding runs by making a double tree of hubs, alongwith every hub which is a leaf hub or an inner hub. All hubs are at first leaf hubs, and there is 1 leaf hub for every character in message being packed. Leaf hub contains the recurrence and the character of utilization for that particular character. Internal hubs comprises connections of 2 youngster hubs in addition to a recurrence that is entirety of frequencies of 2 tyke hubs.

Path lengths are distinctive : The tree is built with the end goal that the ways from the root to the most oftentimes utilized characters are brief time the ways to less much of the time utilized characters are very lengthy. This outcomes in short program for regularly utilized characters and long codes for less much of the time utilized characters.

Make a recurrence diagram: Invoke the create Freq Data strategy to make a recurrence table that distinguishes every one of the individual characters in the first message and the circumstances (frequency)that each character shows up in the message being compacted.

Make the leaves and build the tree: Invoke the create Leaves strategy to make a Huff Leaf question for every character distinguished in the recurrence table.

At the point when the create Huff Tree strategy restores, the Huff Tree protest will stay as the main question put away in the Tree Set question that beforehand has the majority of the Huff Leaf objects. This is on account of the greater part of the Huff Leaf articles will join with Huff Node items to shape the tree. At the point when two Huff Leaf objects are joined with a solitary Huff Node question, the 2 Huff Leaf elements are expelled from Tree Set protest, and the Huff Node question is added to the Tree Set protest.

Make the bit codes: conjure the create Bit Codes strategy.The pressure gave by Huffman encoding relies upon the much of the time utilized characters that have small piece programs and short often times utilized characters have larger piece programs.

Following are the primary advances of algorithm for pressure and decompression.

StepI: Input thetext data to be compressed.

StepII: Apply Dynamic bitReductionmethod to compress the data.

StepIII: Find the unique symboltocompress the data further.

StepIV: Create the binarytree with nodes representing theunique symbols

Step V: ApplyHuffman coding to Finallycompress the data.

Step VI:Display the final result obtained in previous step.

SYMBOL	CODE
A1	0
A2	10
A3	111
A4	110

At first, all hubs are leaf hubs, which contain the image itself, the weight of the image and alternatively, Link to a parent hub which makes it simple to read the code (inreverse)starting from a leaf hub. Internal nodes contain image weight, joins to two tyke hubs and the discretionary link to a parent node. A completed treehas Leaf hubs and N−1 internal nodes. A direct time technique to make a Huffman tree is to utilize two lines, the first containing the underlying weights, and joined weights being returned in the of the second line. Most minimal weight is constantly kept at the front.

3.3 DEFLATE ALGORITHM

Deflate algorithm is a lossless data compression scheme which uses combination of LZ77 and Huffman Coding. It is a popular compression algorithm used by gzip zlib. The algorithm searches for duplicate strings in the input data, and then every subsequent occurrence of that duplicate string is replaced by pointer to the previous occurrence of the string. Similar the LZ77 algorithm step, the pointer to previous occurrence of the string is in the form of a pair.In general, distances are upper limited by 32K bytes. However, the lengths are usually limited to 258 bytes. Duplicated strings are found using hash table and each and every input string of length equals to 3 gets embeded into this hash table.

The compression of matchingthe lengths is achieved via one Huffman tree, whereas the compression in match distances is achieved via another Huffman tree. Typically, these trees are kept in packed form at beginning of every block, and they may have any size.

The process of lazy evaluation is done at the runtime using a runtime parameter.

- In case if compression ratio is preferred over execution speed, the algorithm attempts a complete second search for finding a longer match even if the earlier result string is long enough.
- For faster compression speed, the lazy match evaluation is not performed. For such fast modes, the new strings are inserted into the hash table only if no match is found or when the match not long enough.
- If the current match is long enough, then the algorithm reduces search for a longer match.

The decompression algorithm used by deflate algorithm is called inflate algorithm and is discussed next:

3.5 Run-length Encoding (RLE)

RLE is a very easy form of a lossless compression in which *runs* of data, sequences in which the same data value occurs in many consecutive data elements) are stored as a single data value and count, rather than as the original run. This is most useful on data contains many such runs.

RLE is suitedfor compressing any type ofdata regardless of its informationcontent, but the contentofthe data will affect thecompression ratio attained byRLE.Although most RLE algorithms cannot achieve the highcompression ratios ofthemore advanced compression methods, RLE is both easy to implement andquickto execute, makingit a good alternative toeitherusinga complex compression algorithm or leavingyour image data uncompressed. The first byte represents thenumber of characters in the run and is called the *run count*. In practice, an encoded run may contain 1 to 128 or 256 characters; the runtcount usually contains as the number of characters minus one (a value in the range of 0 to 127 or 255).

3.6 LZW Algorithm

LZW pressure is pressure of file into a shorter document using a table-based query calculation developed by Jacob Ziv. Abraham Lempel and Terry Welch. Two ordinarily utilized document arranges in which LZV pressure is utilized are the GIF picture organize served from Web destinations and TIFF imagen design. LZW pressure is additionally reasonable for packing content documents. A specific LZW pressure calculation takes each info succession of bits of a given length and makes a section in a table for that specific piece design, comprising of the example itself and shorter code. As info is perused, any pattern that is perused before brings about substitution of the little code, viably packing the aggregate sum of input to something littler. Table of codes as a major aspect of the compacted record. The n translating program that decompress the document can assemble the table itself by utilizing the calculation as it forms the encoded input.

3.7 Shannon-Fano Coding Algorithm

In Shannon– Fano coding, the images are orchestrated all together from most likely to minimum plausible, and after that separated into two sets whose aggregate probabilities are as close as conceivable to being second. For whatever length of time that any sets with in excess one part remain, a similar procedure is rehashed on those sets, to decide progressive digits of their codes. The calculation delivers genuinely proficient variable-length encodings; when the two littler sets created by a dividing are in actuality of equivalent likelihood, the one piece of data used to recognize them is utilized generally effectively. Lamentably, Shannon– Fano does not generally deliver ideal prefix codes; the arrangement of probabilities {0.35, 0.17, 0.17, 0.16, 0.15} is a case of one that will be allocated non-ideal codes by Shannon– Fano coding. Thus, Shannon– Fano is never utilized; Huffman coding is nearly as computationally basic and produces prefix codes that dependably accomplish the most minimal expected code word length, under the limitations that every image is spoken to by a code shaped of a basic number of bits.

3.8 Burrows-Wheeler Algorithm

The BWT is a data transformation algorithm that restructures data in such a way that the transformed message is more compressible. Technically, it is lexicographical reversible permutation of the characters of a string. It is first of the three steps to be performed in succession while implementing Burrows – Wheeler Data Compression algorithm that forms the basis of the Unix compression utility bzip2. The most important application of BWT is found in biological sciences where genomes don’t have many runs but they do have many repeats.

The idea of the BWT is to build an array whose rows are all cyclic shifts of the input string in dictionary order, and return the last column of the array that tends to have long runs of identical characters. The benefit of this is that once the characters have been clustered together, they effectively have an ordering, which can make our string more compressible for other algorithms like run length encoding and Huffman Coding.Time Complexity:O(log n) This is because of the method used above to build suffix array which has O(Log n) time complexity, due to O(n) time for strings comparisons in O(nLogn) sorting algorithm.

3.9 Arithmetic Encoding Algorithm

Arithmetic coding is a typical calculation utilized as a part of both lossless and lossy information pressure calculations. It is an entropy encoding strategy, in which the as often as possible seen images are encoded with less bits than once in a while observed images. It has a few points of interest over surely understood procedures like Huffman coding. It changes over whole info information into a solitary gliding point number n where ($0.0 <= n < 1.0$).The interim is isolated into sub-interims in the proportion of the likelihood of event frequencies.For a startpoint and endpoint of a whole range the lower-furthest reaches of a character go is the maximum furthest reaches of the past character given by startpoint + combined recurrence X (endpoint - startpoint).

CHAPTER IV

PERFORMANCE ANALYSIS AND RESULTS

Several factors may be used to study the performance analysis of the compression algorithms. However, space efficiency and time efficiency plays important role in study of algorithms. For data compression algorithms, another factor called compression ratio also known as compression power is used to represent the relative figure of merit of algorithms.

4.1 COMPRESSION RATIO

Compression ratio of data compression ratio is define as ratio between uncompressed data size and the compressed data size as below :

$$\text{Compression Ratio} = \frac{\text{Uncompressed Data Size}}{\text{Compressed Data Size}}$$

The compression ratio is a measure analogous to the physical compression ratio in mechanical systems of substances. Lesser the size of the compressed data, larger will be the compression ratio.

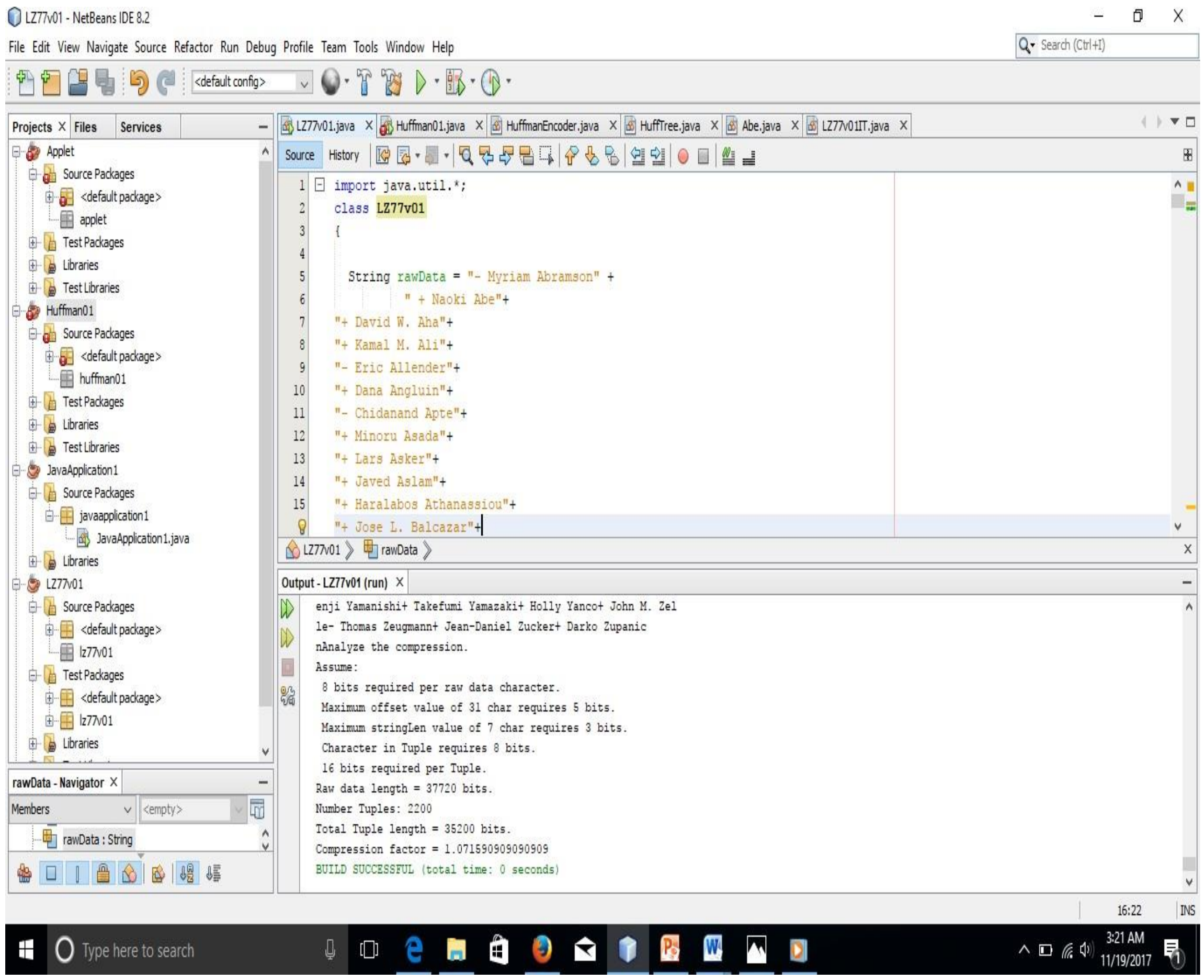
4.2 COMPRESSION SPEED

Compression speed is related to a particular data format and the type of . There exists a relationship between the compression speed / application performance vs the host machine parameters. However, machine specific studies have not been done in this work. During this project work, the same machine is used for implementation of all the compression algorithms.

$$\text{Speed} = \frac{\text{Uncompressed Bits}}{\text{speed to compress}}$$

4.3 PERFORMANCE ANALYSIS OF DATA COMPRESSION ALGORITHMS

4.3.1 LZ77 Program & Test Case Output


```java
import java.util.*;
class LZ77v01
{

  String rawData = "- Myriam Abramson" +
          " + Naoki Abe"+
"+ David W. Aha"+
"+ Kamal M. Ali"+
"- Eric Allender"+
"+ Dana Angluin"+
"- Chidanand Apte"+
"+ Minoru Asada"+
"+ Lars Asker"+
"+ Javed Aslam"+
"+ Haralabos Athanassiou"+
"+ Jose L. Balcazar"+
```

```
enji Yamanishi+ Takefumi Yamazaki+ Holly Yanco+ John M. Zel
le- Thomas Zeugmann+ Jean-Daniel Zucker+ Darko Zupanic
nAnalyze the compression.
Assume:
 8 bits required per raw data character.
 Maximum offset value of 31 char requires 5 bits.
 Maximum stringLen value of 7 char requires 3 bits.
 Character in Tuple requires 8 bits.
 16 bits required per Tuple.
Raw data length = 37720 bits.
Number Tuples: 2200
Total Tuple length = 35200 bits.
Compression factor = 1.071590909090909
BUILD SUCCESSFUL (total time: 0 seconds)
```

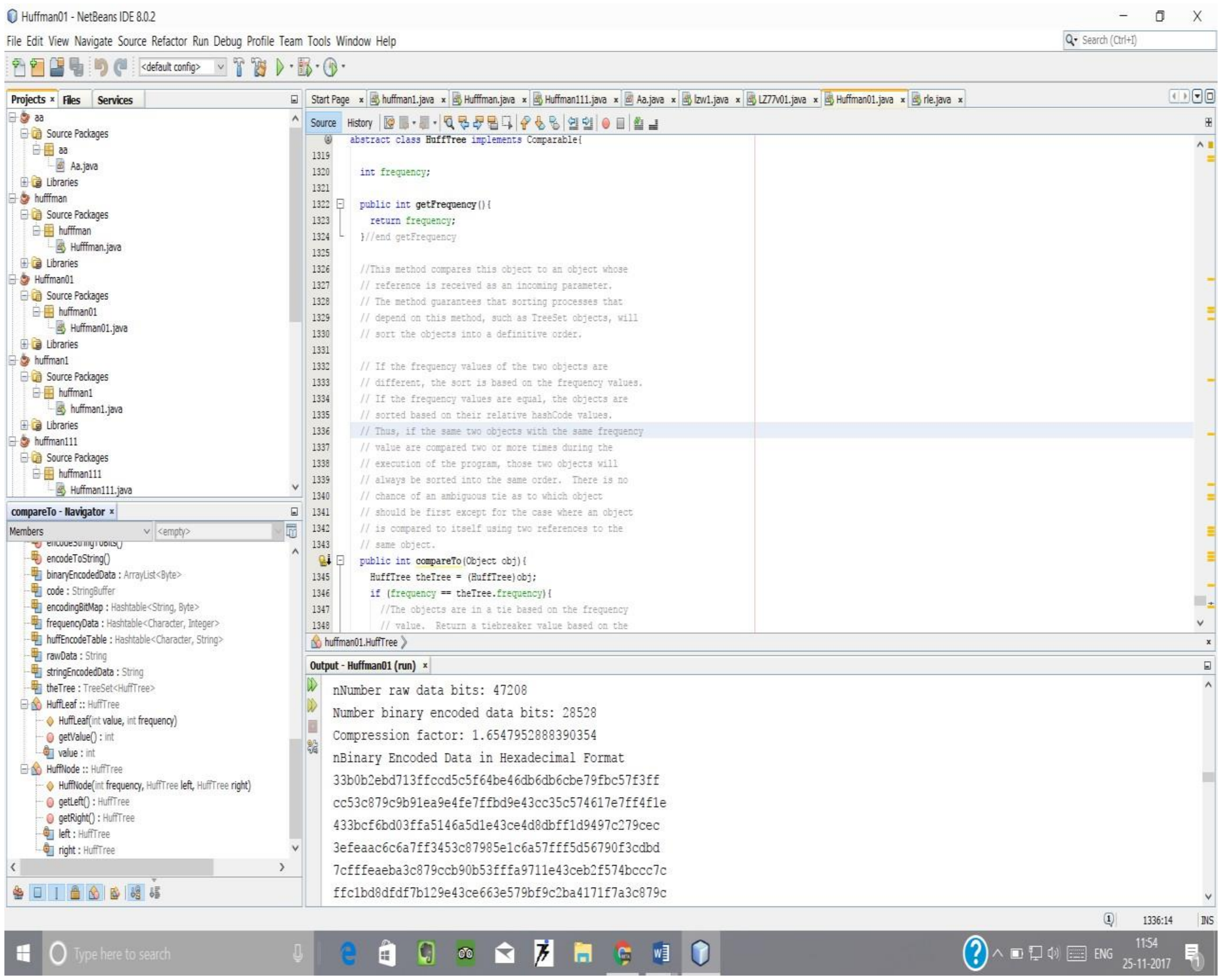
Fig 4.1: LZ77 Program & Test Case Output

Analyze the compression.

Compression factor = 1.071590909090909

BUILD SUCCESSFUL (total time: 0 seconds)

4.3.2 HuffmanCodingProgramOutput



```java
abstract class HuffTree implements Comparable{

  int frequency;

  public int getFrequency(){
    return frequency;
  }//end getFrequency

  //This method compares this object to an object whose
  // reference is received as an incoming parameter.
  // The method guarantees that sorting processes that
  // depend on this method, such as TreeSet objects, will
  // sort the objects into a definitive order.

  // If the frequency values of the two objects are
  // different, the sort is based on the frequency values.
  // If the frequency values are equal, the objects are
  // sorted based on their relative hashCode values.
  // Thus, if the same two objects with the same frequency
  // value are compared two or more times during the
  // execution of the program, those two objects will
  // always be sorted into the same order.  There is no
  // chance of an ambiguous tie as to which object
  // should be first except for the case where an object
  // is compared to itself using two references to the
  // same object.
  public int compareTo(Object obj){
    HuffTree theTree = (HuffTree)obj;
    if (frequency == theTree.frequency){
      //The objects are in a tie based on the frequency
      // value.  Return a tiebreaker value based on the
```

```
nNumber raw data bits: 47208
Number binary encoded data bits: 28528
Compression factor: 1.6547952888390354
nBinary Encoded Data in Hexadecimal Format
33b0b2ebd713ffccd5c5f64be46db6db6cbe79fbc57f3ff
cc53c879c9b91ea9e4fe7ffbd9e43cc35c574617e7ff4f1e
433bcf6bd03ffa5146a5d1e43ce4d8dbff1d9497c279cec
3efeaac6c6a7ff3453c87985e1c6a57fff5d56790f3cdbd
7cfffeaeba3c879ccb90b53fffa9711e43ceb2f574bccc7c
ffc1bd8dfdf7b129e43ce663e579bf9c2ba4171f7a3c879c
```

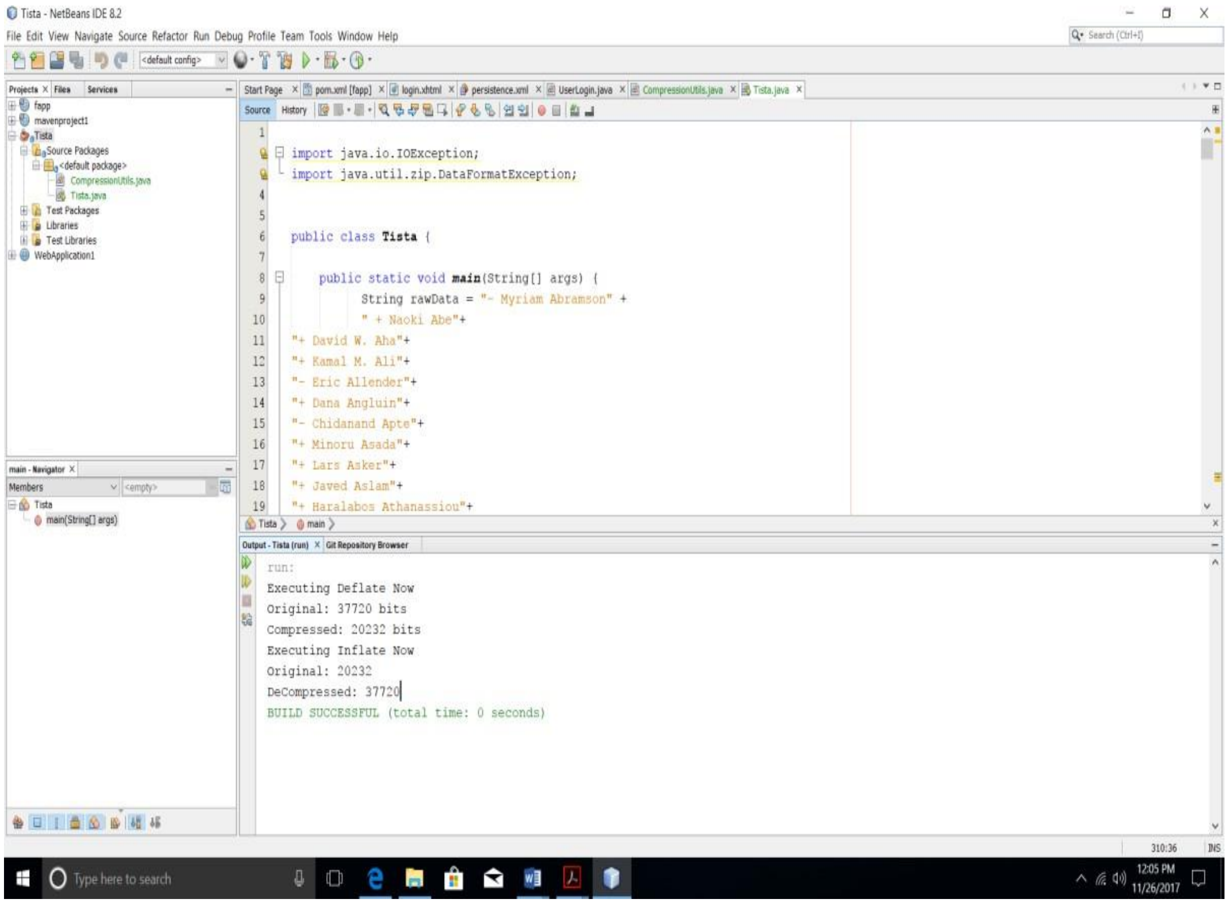
Fig 4.2 Huffman Coding Program Output

Number of Raw Bits : 47208

Number of bits in encoded string: 28528

Compression Ratio: 1.6547

4.3.3 Deflate and Inflate Algorithm Output



```java

import java.io.IOException;
import java.util.zip.DataFormatException;


public class Tista {

    public static void main(String[] args) {
          String rawData = "- Myriam Abramson" +
          " + Naoki Abe"+
"+ David W. Aha"+
"+ Kamal M. Ali"+
"- Eric Allender"+
"+ Dana Angluin"+
"- Chidanand Apte"+
"+ Minoru Asada"+
"+ Lars Asker"+
"+ Javed Aslam"+
"+ Haralabos Athanassiou"+
```

```
run:
Executing Deflate Now
Original: 37720 bits
Compressed: 20232 bits
Executing Inflate Now
Original: 20232
DeCompressed: 37720
BUILD SUCCESSFUL (total time: 0 seconds)
```

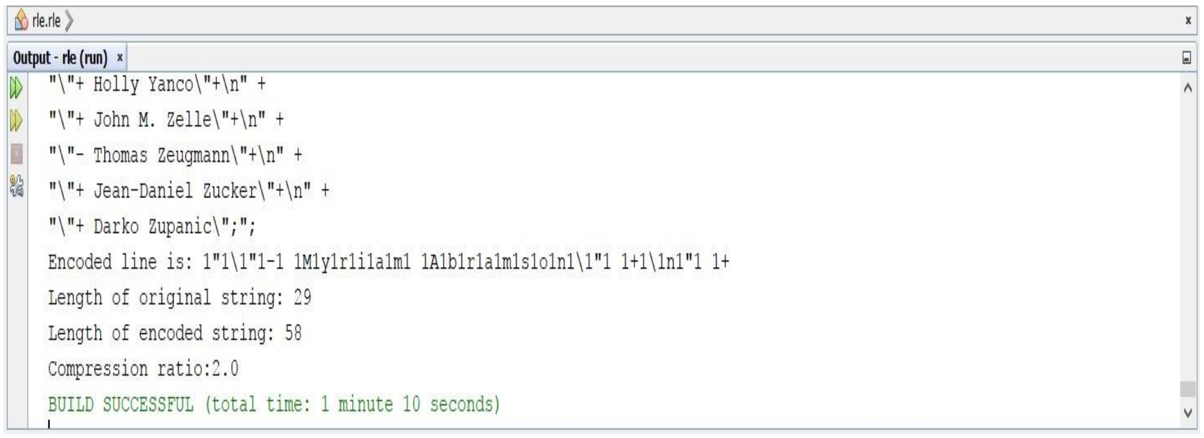
Fig 4.3 Deflate and Inflate Algorithm Output

Length of the original string : 37720 bits

Length of the Compressed String: 20232 bits

Compression Ratio: 1.864

4.3.4 Run Length Coding Program Output

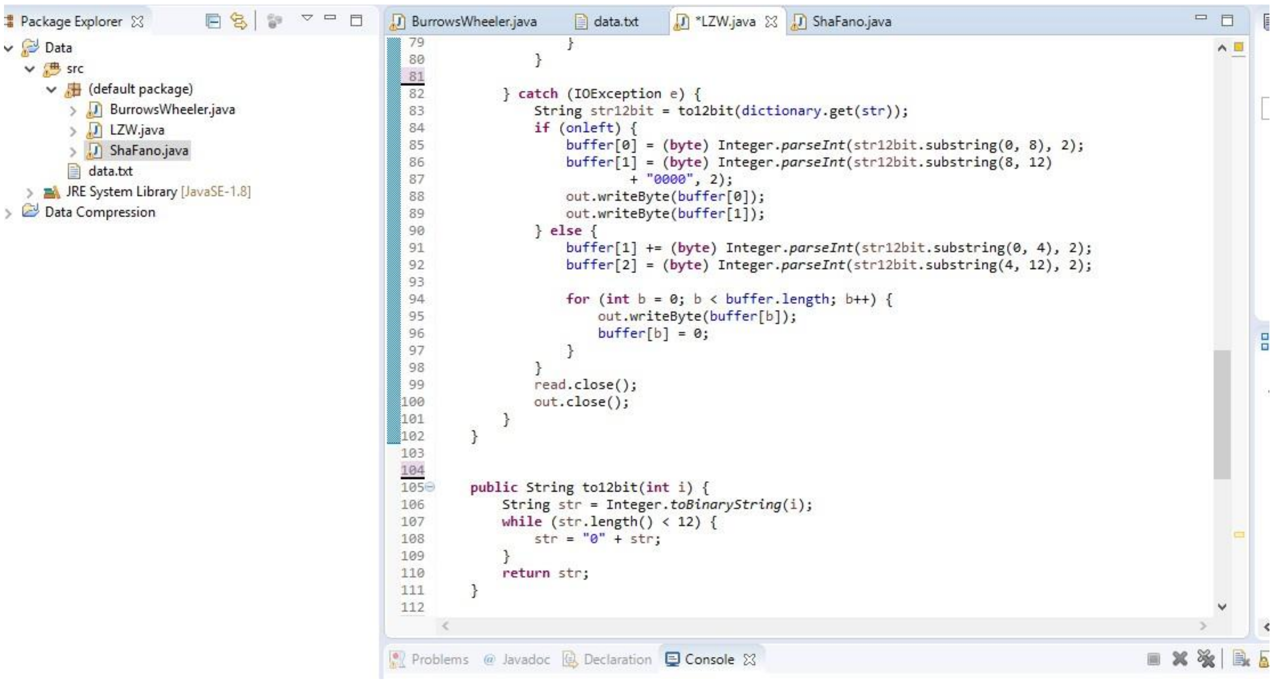


```
"\"+ Holly Yanco\"+\n" +
"\"+ John M. Zelle\"+\n" +
"\"- Thomas Zeugmann\"+\n" +
"\"+ Jean-Daniel Zucker\"+\n" +
"\"+ Darko Zupanic\";";
Encoded line is: 1"1\1"1-1 1M1y1r1i1a1m1 1A1b1r1a1m1s1o1n1\1"1 1+1\1n1"1 1+
Length of original string: 29
Length of encoded string: 58
Compression ratio:2.0
BUILD SUCCESSFUL (total time: 1 minute 10 seconds)
```

Fig 4.4 : Run Length Coding Program Output

1. Length of the original string: 29
2. Length of the original string: 58
3. Compression ratio: 2.0

4.3.5 LZW Program & Test Case Output



```java
                }
            }

        } catch (IOException e) {
            String str12bit = to12bit(dictionary.get(str));
            if (onleft) {
                buffer[0] = (byte) Integer.parseInt(str12bit.substring(0, 8), 2);
                buffer[1] = (byte) Integer.parseInt(str12bit.substring(8, 12)
                        + "0000", 2);
                out.writeByte(buffer[0]);
                out.writeByte(buffer[1]);
            } else {
                buffer[1] += (byte) Integer.parseInt(str12bit.substring(0, 4), 2);
                buffer[2] = (byte) Integer.parseInt(str12bit.substring(4, 12), 2);

                for (int b = 0; b < buffer.length; b++) {
                    out.writeByte(buffer[b]);
                    buffer[b] = 0;
                }
            }
            read.close();
            out.close();
        }
    }


    public String to12bit(int i) {
        String str = Integer.toBinaryString(i);
        while (str.length() < 12) {
            str = "0" + str;
        }
        return str;
    }

```

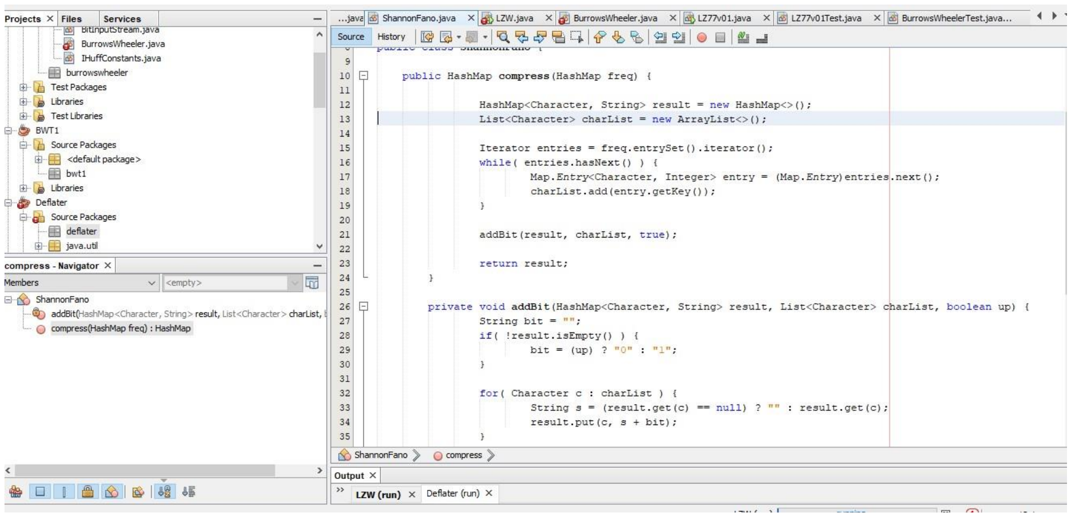
Fig 4.5 : LZW Program & Test Case Output

Number of Raw Bits : 47208

Number of bits in encoded string: 32777

Compression Ratio- 2.9

4.3.6 Shannon-Fano Program & Test Case Output



```
    public HashMap compress(HashMap freq) {

            HashMap<Character, String> result = new HashMap<>();
            List<Character> charList = new ArrayList<>();

            Iterator entries = freq.entrySet().iterator();
            while( entries.hasNext() ) {
                    Map.Entry<Character, Integer> entry = (Map.Entry)entries.next();
                    charList.add(entry.getKey());
            }

            addBit(result, charList, true);

            return result;
        }

        private void addBit(HashMap<Character, String> result, List<Character> charList, boolean up) {
            String bit = "";
            if( !result.isEmpty() ) {
                    bit = (up) ? "0" : "1";
            }

            for( Character c : charList ) {
                    String s = (result.get(c) == null) ? "" : result.get(c);
                    result.put(c, s + bit);
            }
```

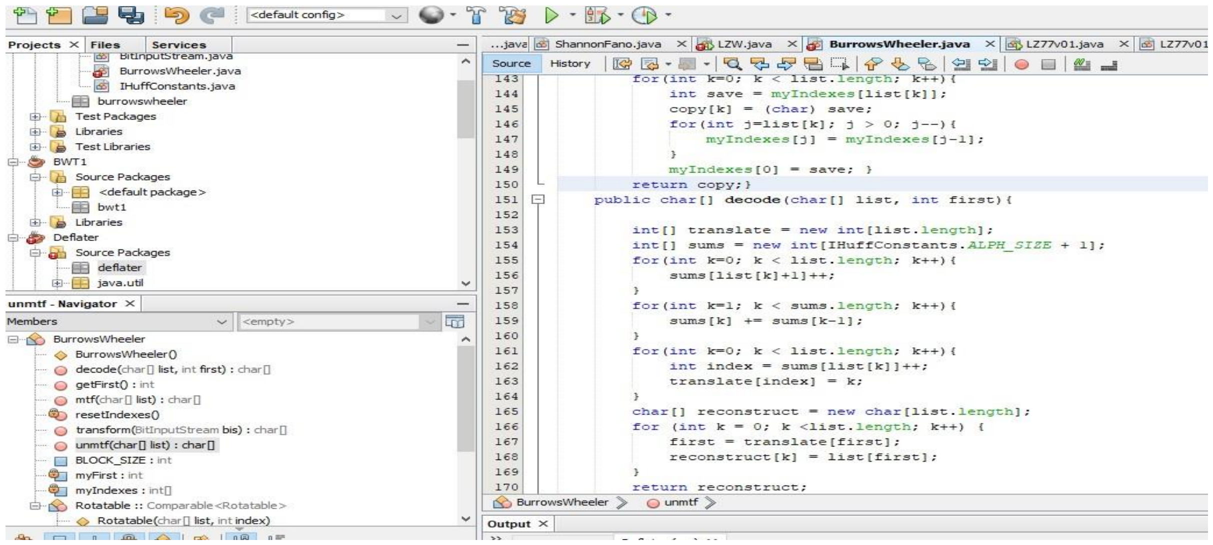
Fig 4.6: Shannon-Fano Program & Test Case Output

Number of Raw Bits : 47208

Number of bits in encoded string: 38798

Compression Ratio- 2.46

4.3.7 Burrows-Wheeler Program & Test Case Output



```
            for(int k=0; k < list.length; k++){
                int save = myIndexes[list[k]];
                copy[k] = (char) save;
                for(int j=list[k]; j > 0; j--){
                    myIndexes[j] = myIndexes[j-1];
                }
                myIndexes[0] = save; }
            return copy;}
        public char[] decode(char[] list, int first){

            int[] translate = new int[list.length];
            int[] sums = new int[IHuffConstants.ALPH_SIZE + 1];
            for(int k=0; k < list.length; k++){
                sums[list[k]+1]++;
            }
            for(int k=1; k < sums.length; k++){
                sums[k] += sums[k-1];
            }
            for(int k=0; k < list.length; k++){
                int index = sums[list[k]]++;
                translate[index] = k;
            }
            char[] reconstruct = new char[list.length];
            for (int k = 0; k <list.length; k++) {
                first = translate[first];
                reconstruct[k] = list[first];
            }
            return reconstruct;
```

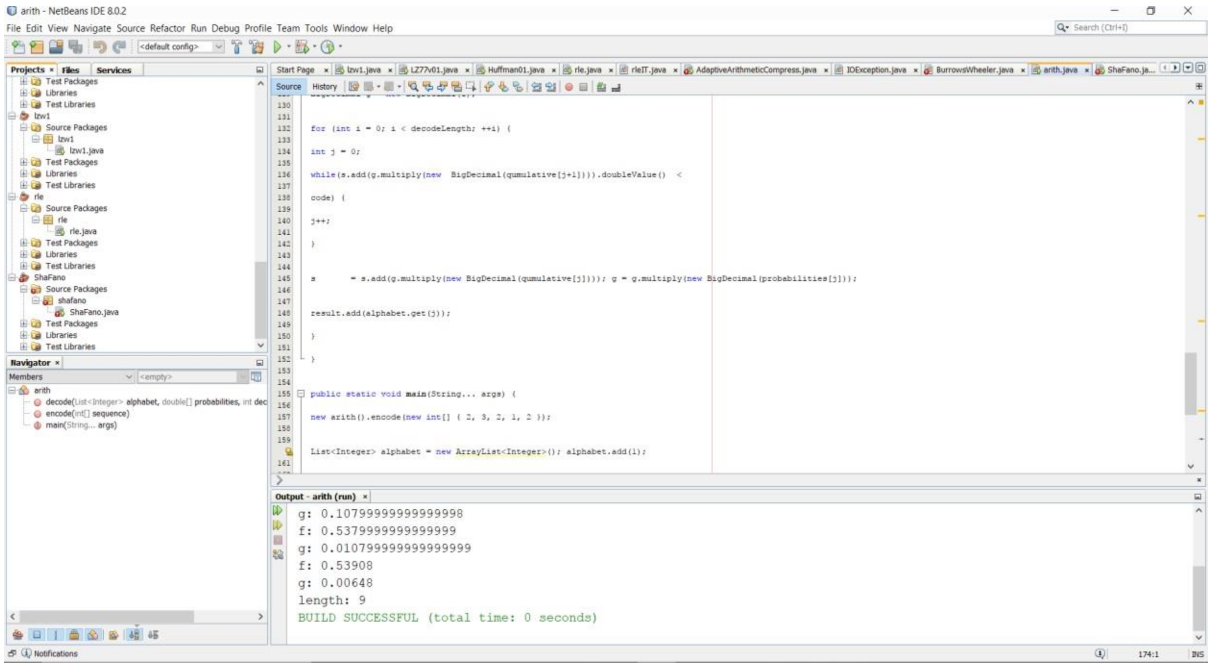
Fig 4.7: Burrows-Wheeler Program & Test Case Output

Number of Raw Bits : 47208

Number of bits in encoded string: 29928

Compression Ratio- 0.434

4.3.8 Arithmetic Encoding Program & Test Case Output



```
for (int i = 0; i < decodeLength; ++i) {

int j = 0;

while(s.add(g.multiply(new  BigDecimal(qumulative[j+1]))).doubleValue()  <

code) {

j++;

}


s       = s.add(g.multiply(new BigDecimal(qumulative[j]))); g = g.multiply(new BigDecimal(probabilities[j]));


result.add(alphabet.get(j));

}

}


public static void main(String... args) {

new arith().encode(new int[] { 2, 3, 2, 1, 2 });


List<Integer> alphabet = new ArrayList<Integer>(); alphabet.add(1);
```

```
g: 0.10799999999999998
f: 0.5379999999999999
g: 0.010799999999999999
f: 0.53908
g: 0.00648
length: 9
BUILD SUCCESSFUL (total time: 0 seconds)
```

Fig 4.8: Arithmetic Encoding Program & Test Case Output

Number of Raw Bits : 47208

Number of bits in encoded string: 27628

Compression Ratio- 0.99

4.4 PERFORMANCE COMPARISON OF ALGORITHMS

The compression ratio of the following implemented algorithm was studied.

- LZ77
- Huffman Coding
- DEFLATE Algorithm
- Run Length Coding
- Arithmetic Encoding
- Shannon-Fano Encoding
- LZW Algorithm
- Burrows-Wheeler Algorithm

The relative performance measure in terms of compression ratio are listed in the Table 4.1

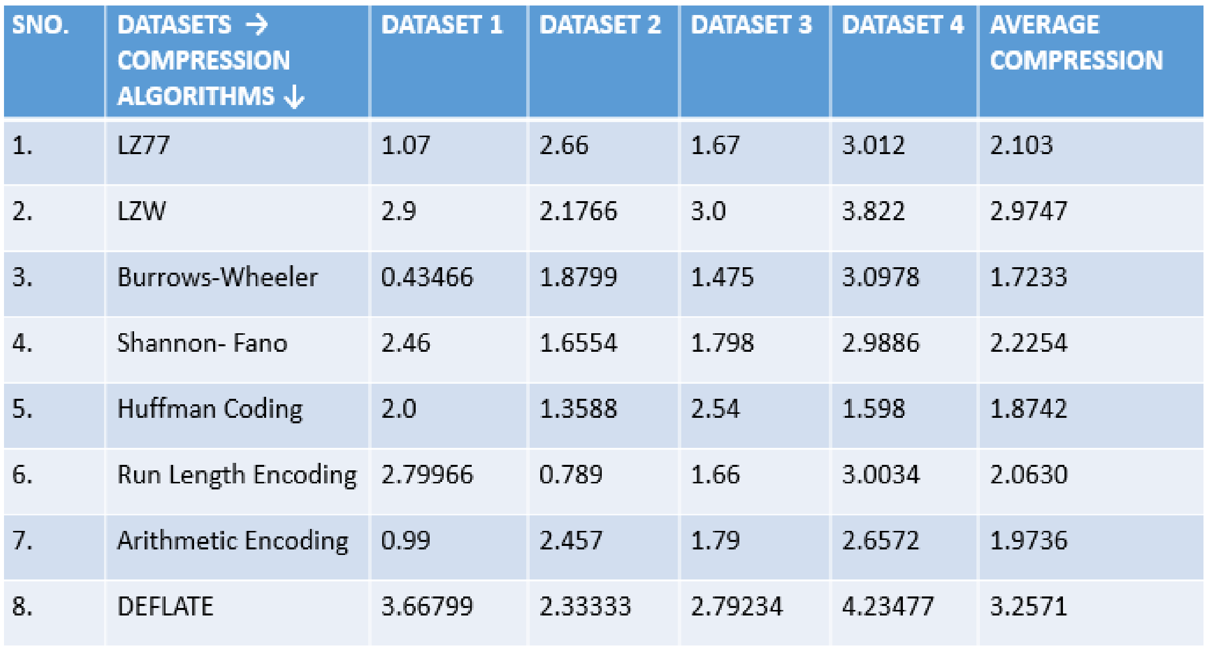
SNO.	DATASETS → COMPRESSION ALGORITHMS ↓	DATASET 1	DATASET 2	DATASET 3	DATASET 4	AVERAGE COMPRESSION
1.	LZ77	1.07	2.66	1.67	3.012	2.103
2.	LZW	2.9	2.1766	3.0	3.822	2.9747
3.	Burrows-Wheeler	0.43466	1.8799	1.475	3.0978	1.7233
4.	Shannon- Fano	2.46	1.6554	1.798	2.9886	2.2254
5.	Huffman Coding	2.0	1.3588	2.54	1.598	1.8742
6.	Run Length Encoding	2.79966	0.789	1.66	3.0034	2.0630
7.	Arithmetic Encoding	0.99	2.457	1.79	2.6572	1.9736
8.	DEFLATE	3.66799	2.33333	2.79234	4.23477	3.2571

From the table, it is clear that the Deflate algorithm's performance is relatively better as compared to other algorithms followed by algorithms from LZ family i.e. LZW and LZ77 algorithms with 2.9747 and 2.103 compression ratios respectively. However, best compression is given by DEFLATE algorithm i.e 3.2571.

CHAPTER VI

CONCLUSIONS

Lossless data compression techniques have been studied during this project. The studies started with brief literature review of the state of the art algorithms in the area of data compression. Lossless data compression algorithms were selected for this study. These algorithms are particularly used for text and data compression due to exactly recovery of text/data. Further, these algorithms also find applications in medical imaging etc. Eight algorithms,namely: LZ77, Huffman Coding, Deflate algorithm and Run Length coding,Arithmetic encoding, Burrows-Wheeler Transform, Shannon-Fano encoding and LZW algorithm were studied as a part of the project. All of these algorithms were implemented using programming languages Java and C++ and common string was chosen as input string so as to compare these algorithms.



SNO.	DATASETS → COMPRESSION ALGORITHMS ↓	DATASET 1	DATASET 2	DATASET 3	DATASET 4	AVERAGE COMPRESSION
1.	LZ77	1.07	2.66	1.67	3.012	2.103
2.	LZW	2.9	2.1766	3.0	3.822	2.9747
3.	Burrows-Wheeler	0.43466	1.8799	1.475	3.0978	1.7233
4.	Shannon- Fano	2.46	1.6554	1.798	2.9886	2.2254
5.	Huffman Coding	2.0	1.3588	2.54	1.598	1.8742
6.	Run Length Encoding	2.79966	0.789	1.66	3.0034	2.0630
7.	Arithmetic Encoding	0.99	2.457	1.79	2.6572	1.9736
8.	DEFLATE	3.66799	2.33333	2.79234	4.23477	3.2571

From the results it is clear that deflate algorithm followed by the algorithm LZ family likewise LZW and LZ77 performs the best in terms of compression ratio.

REFERENCES

1. Khalid Sayood, “Introduction to Data Compression”, 4th Edition, Morgan Kaufmann Publishing, MA, 2012.
2. D. Salomon, “Data Compression: The Complete Reference”, 3rd Edition, Springer-Verlag, New York, 2004.
3. D. A. Huffman, “A Method for the Construction of Minimum Redundancy Codes”, Proceedings of the Institute of Radio Engineers, vol. 40, no. 9, pp. 1098-1101, 1952.
4. R. Pasco,” Source coding algorithms for fast data compression,” Stanford Univ., Ph.D. Dissertation, May 1976.
5. J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression", IEEE Transactions on Information Theory, vol. 23, no. 3, pp. 337-343, 1977.
6. J. Ziv and A. Lempel,“Compression of individual sequences via variable-ratecoding”, IEEETransactionsonInformationTheory, vol. 24, 530–536, 1978.
7. M. Burrows and D. J. Wheeler. “A Block–sorting Lossless Data Compression Algorithm,” SRC Research Report 124, Digital Systems Research Center, Palo Alto, CA, May 1994.
8. C.E. Shannon, “A mathematical theory of communication”, The Bell Syst. Tech. Jr., Vol. 27, pp. 379–423, 623–656, July, October, 1948.
9. Glen G. Langdon, Jr, “An Introduction to Arithmetic Coding.” IBM Journal of Research and Development, vol. 28, no. 2, pp. 135-149, March 1984.
10. P. G. Howard and J. S. Vitter, “Arithmetic coding for data compression,” Proceedings of IEEE, vol. 82, no. 6, pp. 857—865, June 1994.
11. P. Deutsch, DEFLATE Compressed Data Format, Aladdin Enterprises Category: May 1996. RFC 1951.
12. M. Guazoo, “A general minimum-redundancy source-coding algorithm,” IEEE Trans. on Information Theory, vol. IT-26, no. 1, pp. 15-25, Jan 1980.
13. M. Hosseini, D. Pratas and A. J. PinhoA Survey on Data Compression Methods for Biological Sequences Information 2016, 7, 56; doi:10.3390/info7040056.
14. N. Kimura and S. Latifi, “ A Survey on Data Compression in Wireless Sensor Networks”, Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC’05) IEEE.
15. G. Sethi, S. Shaw, K. Vinutha and C. Chakravorty, “Data Compression Techniques”, Int JR. of Comp Sc. And Information Tech., vol. 5, no. 4, pp-5584-5586, 2014
16. S. Kaur, V. S. Verma, “Design and Implementation of LZW Data Compression Algorithm”, Inr. Jr. of Inf. Sciences and Techniques, vol. 2, no. 4, pp. 71-81, 2012.

Web References:

http://www.stringology.org/DataCompression/ak-int/index_en.html

https://en.wikipedia.org/wiki/LZ77_and_LZ78
https://en.wikipedia.org/wiki/DEFLATE
http://cotty.16x16.com/compress/nelson1.htm
http://www.drdobbs.com/parallel/arithmetic-coding-and-statistical-modeli/184408491
https://www.cs.duke.edu/csed/curious/compression/lzw.html

APPENDIX I
DATASET AND SOURCE CODE

DATASET CHOSEN FOR COMPRESSION

```
String rawData = "\"- Myriam Abramson\" +\n" +
"          \" + Naoki Abe\"+\n" +
"\"+ David W. Aha\"+\n" +
"\"+ Kamal M. Ali\"+\n" +
"\"- Eric Allender\"+\n" +
"\"+ Dana Angluin\"+\n" +
"\"- Chidanand Apte\"+\n" +
"\"+ Minoru Asada\"+\n" +
"\"+ Lars Asker\"+\n" +
"\"+ Javed Aslam\"+\n" +
"\"+ Haralabos Athanassiou\"+\n" +
"\"+ Jose L. Balcazar\"+\n" +
"\"+ Timothy P. Barber\"+\n" +
"\"+ Michael W. Barley\"+\n" +
"\"- Cristina Baroglio\"+\n" +
"\"+ Peter Bartlett\"+\n" +
"\"- Eric Baum\"+\n"  +
"\"+ Welton Becket\"+\n" +
"\"- Shai Ben-David\"+\n" +
"\"+ George Berg\"+\n" +
"\"+ Neil Berkman\"+\n" +
"\"+ Malini Bhandaru\"+\n" +
"\"+ Bir Bhanu\"+\n" +
"\"+ Reinhard Blasig\"+\n" +
"\"- Avrim Blum\"+\n" +
"\"- Anselm Blumer\"+\n" +
"\"+ Justin Boyan\"+\n" +
"\"+ Carla E. Brodley\"+\n" +
"\"+ Nader Bshouty\"+\n" +
"\"- Wray Buntine\"+\n" +
"\"- Andrey Burago\"+\n" +
"\"+ Tom Bylander\"+\n" +
"\"+ Bill Byrne\"+\n" +
"\"- Claire Cardie\"+\n"  +
"\"+ Richard A. Caruana\"+\n" +
"\"+ John Case\"+\n" +
"\"+ Jason Catlett\"+\n" +
"\"+ Nicolo Cesa-Bianchi\"+\n" +
"\"- Philip Chan\"+\n" +
"\"+ Mark Changizi\"+\n" +
"\"+ Pang-Chieh Chen\"+\n" +
"\"- Zhixiang Chen\"+\n" +
"\"+ Wan P. Chiang\"+\n" +
"\"- Steve A. Chien\"+\n" +
"\"+ Jeffery Clouse\"+\n" +
"\"+ William Cohen\"+\n" +
"\"+ David Cohn\"+\n" +
"\"- Clare Bates Congdon\"+\n" +
"\"- Antoine Cornuejols\"+\n" +
"\"+ Mark W. Craven\"+\n" +
"\"+ Robert P. Daley\"+\n" +
"\"+ Lindley Darden\"+\n" +
"\"- Chris Darken\"+\n" +
"\"- Bhaskar Dasgupta\"+\n" +
"\"- Brian D. Davidson\"+\n" +
"\"+ Michael de la Maza\"+\n" +
"\"- Olivier De Vel\"+\n" +
"\"- Scott E. Decatur\"+\n" +
"\"+ Gerald F. DeJong\"+\n" +
"\"+ Kan Deng\"+\n" +
"\"- Thomas G. Dietterich\"+\n" +
"\"+ Michael J. Donahue\"+\n" +
"\"+ George A. Drastal\"+\n" +
"\"+ Harris Drucker\"+\n" +
"\"- Chris Drummond\"+\n" +
"\"+ Hal Duncan\"+\n" +
"\"- Thomas Ellman\"+\n" +
"\"+ Tapio Elomaa\"+\n" +
"\"+ Susan L. Epstein\"+\n" +
"\"+ Bob Evans\"+\n" +
"\"- Claudio Facchinetti\"+\n" +
"\"+ Tom Fawcett\"+\n" +
"\"- Usama Fayyad\"+\n" +
"\"+ Aaron Feigelson\"+\n" +
"\"+ Nicolas Fiechter\"+\n" +
"\"+ David Finton\"+\n" +
"\"+ John Fischer\"+\n" +
"\"+ Paul Fischer\"+\n" +
"\"+ Seth Flanders\"+\n" +
"\"+ Lance Fortnow\"+\n" +
"\"- Ameur Foued\"+\n"  +
"\"+ Judy A. Franklin\"+\n" +
"\"+ Yoav Freund\"+\n" +
"\"+ Johannes Furnkranz\"+\n" +
"\"+ Merrick L. Furst\"+\n" +
"\"+ Jean Gabriel Ganascia\"+\n" +
"\"+ William Gasarch\"+\n" +
"\"+ Ricard Gavalda\"+\n" +
"\"+ Melinda T. Gervasio+\"+\n" +
"\"+ Yolanda Gil\"+\n" +
"\"+ David Gillman\"+\n" +
"\"- Attilio Giordana\"+\n" +
"\"+ Kate Goelz\"+\n" +
"\"+ Paul W. Goldberg\"+\n" +
"\"+ Sally Goldman\"+\n" +
"\"+ Diana Gordon\"+\n"  +
"\"+ Geoffrey Gordon\"+\n" +
"\"+ Jonathan Gratch\"+\n" +
"\"+ Leslie Grate\"+\n"  +
"\"+ William A. Greene\"+\n" +
"\"+ Russell Greiner\"+\n" +
"\"+ Marko Grobelnik\"+\n" +
"\"+ Tal Grossman\"+\n"  +
"\"+ Margo Guertin\"+\n" +
"\"+ Tom Hancock\"+\n" +
"\"+ Earl S. Harris Jr.\"+\n" +
"\"+ David Haussler\"+\n" +
"\"+ Matthias Heger\"+\n" +
"\"+ Lisa Hellerstein\"+\n" +
"\"+ David Helmbold\"+\n" +
"\"+ Daniel Hennessy\"+\n" +
"\"+ Haym Hirsh\"+\n" +
"\"+ Jonathan Hodgson\"+\n" +
"\"+ Robert C. Holte\"+\n" +
"\"+ Jiarong Hong\"+\n" +
"\"- Chun-Nan Hsu\"+\n" +
"\"+ Kazushi Ikeda\"+\n" +
"\"+ Masayuki Inaba\"+\n" +
"\"- Drago Indjic\"+\n" +
"\"+ Nitin Indurkhya\"+\n" +
"\"+ Jeff Jackson\"+\n" +
"\"+ Sanjay Jain\"+\n"  +
"\"+ Wolfgang Janko\"+\n" +
"\"- Klaus P. Jantke\"+\n" +
"\"+ Nathalie Japkowicz\"+\n" +
"\"+ George H. John\"+\n" +
"\"+ Randolph Jones\"+\n" +
"\"+ Michael I. Jordan\"+\n" +
"\"+ Leslie Pack Kaelbling\"+\n" +
"\"+ Bala Kalyanasundaram\"+\n" +
"\"- Thomas E. Kammeyer\"+\n" +
"\"- Grigoris Karakoulas\"+\n" +
"\"+ Michael Kearns\"+\n" +
"\"+ Neela Khan\"+\n" +
"\"+ Roni Khardon\"+\n" +
"\"+ Dennis F. Kibler\"+\n" +
"\"+ Jorg-Uwe Kietz\"+\n" +
"\"- Efim Kinber\"+\n" +
"\"- Jyrki Kivinen\"+\n" +
"\"- Emanuel Knill\"+\n" +
"\"- Craig Knoblock\"+\n" +
"\"+ Ron Kohavi\"+\n" +
"\"+ Pascal Koiran\"+\n" +
"\"+ Moshe Koppel\"+\n" +
"\"+ Daniel Kortenkamp\"+\n" +
"\"+ Matevz Kovacic\"+\n" +
"\"- Stefan Kramer\"+\n" +
"\"+ Martinch Krikis\"+\n" +
"\"+ Martin Kummer\"+\n" +
"\"- Eyal Kushilevitz\"+\n" +
"\"- Stephen Kwek\"+\n"  +
"\"+ Wai Lam\"+\n" +
"\"+ Ken Lang\"+\n" +
"\"- Steffen Lange\"+\n" +
"\"+ Pat Langley\"+\n" +
"\"+ Mary Soon Lee\"+\n" +
"\"+ Wee Sun Lee\"+\n" +
"\"+ Moshe Leshno\"+\n" +
"\"+ Long-Ji Lin\"+\n" +
"\"- Charles X. Ling\"+\n" +
"\"+ Michael Littman\"+\n" +
"\"+ David Loewenstern\"+\n" +
"\"- Phil Long\"+\n" +
"\"+ Wolfgang Mass\"+\n" +
"\"- Bruce A. MacDonald\"+\n" +
"\"+ Rich Maclin\"+\n" +
"\"- Sridhar Mahadevan\"+\n" +
"\"- J. Jeffrey Mahoney\"+\n" +
"\"+ Yishay Mansour\"+\n" +
"\"+ Mario Marchand\"+\n" +
"\"- Shaul Markovitch\"+\n" +
"\"- Oded Maron\"+\n" +
"\"+ Maja Mataric\"+\n" +
"\"+ David Mathias\"+\n" +
"\"+ Toshiyasu Matsushima\"+\n" +
"\"- Stan Matwin\"+\n" +
"\"- Eddy Mayoraz\"+\n" +
"\"- R. Andrew McCallum\"+\n" +
"\"- L. Thorne McCarty\"+\n" +
"\"- Alexander M. Meystel\"+\n" +
"\"+ Michael A. meystel\"+\n" +
"\"- Steven Minton\"+\n" +
"\"+ Nina Mishra\"+\n"  +
"\"+ Tom M. Mitchell\"+\n" +
"\"+ Dunja Mladenic\"+\n" +
"\"+ David Montgomery\"+\n" +
"\"- Andrew W. Moore\"+\n" +
"\"+ Johanne Morin\"+\n" +
"\"+ Hiroshi Motoda\"+\n" +
"\"- Stephen Muggleton\"+\n" +
"\"+ Patrick M. Murphy\"+\n" +
"\"- Sreerama K. Murthy\"+\n" +
"\"+ Filippo Neri\"+\n" +
"\"- Craig Nevill-Manning\"+\n" +
"\"- Andrew Y. Ng\"+\n" +
"\"+ Nikolay Nikolaev\"+\n" +
"\"- Steven W. Norton\"+\n" +
"\"+ Joseph O'Sullivan\"+\n" +
"\"+ Dan Oblinger\"+\n"  +
"\"+ Jong-Hoon Oh\"+\n" +
"\"- Arlindo Oliveira\"+\n" +
"\"+ David W. Opitz\"+\n" +
"\"+ Sandra Panizza\"+\n" +
"\"+ Barak A. Pearlmutter\"+\n" +
"\"- Ed Pednault\"+\n" +
"\"+ Jing Peng\"+\n" +
"\"+ Fernando Pereira\"+\n" +
"\"+ Aurora Perez\"+\n" +
"\"+ Bernhard Pfahringer\"+\n" +
"\"+ David Pierce\"+\n" +
"\"- Krishnan Pillaipakkamnatt\"+\n" +
"\"+ Roberto Piola\"+\n" +
"\"+ Leonard Pitt\"+\n"  +
"\"+ Lorien Y. Pratt\"+\n" +
"\"- Armand Prieditis\"+\n" +
"\"+ Foster J. Provost\"+\n" +
"\"- J. R. Quinlan\"+\n" +
"\"+ John Rachlin\"+\n"  +
"\"+ Vijay Raghavan\"+\n" +
"\"- R. Bharat Rao\"+\n" +
"\"- Priscilla Rasmussen\"+\n" +
"\"+ Joel Ratsaby\"+\n" +
"\"+ Michael Redmond\"+\n" +
"\"+ Patricia J. Riddle\"+\n" +
"\"+ Lance Riley\"+\n" +
"\"+ Ronald L. Rivest\"+\n" +
"\"+ Huw Roberts\"+\n"  +
"\"+ Dana Ron\"+\n" +
"\"+ Robert S. Roos\"+\n" +
"\"+ Justinian Rosca\"+\n" +
"\"+ John R. Rose\"+\n" +
"\"+ Dan Roth\"+\n" +
"\"+ James S. Royer\"+\n" +
"\"+ Ronitt Rubinfeld\"+\n" +
"\"- Stuart Russell\"+\n" +
"\"+ Lorenza Saitta\"+\n" +
"\"+ Yoshifumi Sakai\"+\n" +
"\"+ William Sakas\"+\n"  +
"\"+ Marcos Salganicoff\"+\n" +
"\"- Steven Salzberg\"+\n" +
"\"- Claude Sammut\"+\n"  +
"\"+ Cullen Schaffer\"+\n" +
"\"+ Robert Schapire\"+\n" +
"\"+ Mark Schwabacher\"+\n" +
"\"+ Michele Sebag\"+\n"  +
"\"+ Gary M. Selzer\"+\n" +
"\"+ Sebastian Seung\"+\n" +
"\"- Arun Sharma\"+\n" +
"\"+ Jude Shavlik\"+\n"  +
"\"+ Daniel L. Silver\"+\n" +
"\"- Glenn Silverstein\"+\n" +
"\"+ Yoram Singer\"+\n"  +
"\"+ Mona Singh\"+\n" +
"\"+ Satinder Pal Singh\"+\n" +
"\"+ Kimmen Sjolander\"+\n" +
"\"+ David B. Skalak\"+\n" +
"\"+ Sean Slattery\"+\n"  +
"\"+ Robert Sloan\"+\n" +
"\"+ Donna Slonim\"+\n" +
"\"+ Carl H. Smith\"+\n" +
"\"+ Sonya Snedecor\"+\n" +
"\"+ Von-Wun Soo\"+\n" +
"\"- Thomas G. Spalthoff\"+\n" +
"\"+ Mark Staley\"+\n" +
"\"- Frank Stephan\"+\n" +
"\"+ Mandayam T. Suraj\"+\n" +
"\"+ Richard S. Sutton\"+\n" +
"\"+ Joe Suzuki\"+\n" +
"\"- Prasad Tadepalli\"+\n" +
"\"+ Hiroshi Tanaka\"+\n" +
"\"- Irina Tchoumatchenko\"+\n" +
"\"- Brian Tester\"+\n" +
"\"- Chen K. Tham\"+\n" +
"\"+ Tatsuo Unemi\"+\n" +
"\"- Lyle H. Ungar\"+\n" +
"\"+ Paul Utgoff\"+\n" +
"\"+ Karsten Verbeurgt\"+\n" +
"\"+ Paul Vitanyi\"+\n"  +
"\"+ Xuemei Wang\"+\n" +
"\"+ Manfred Warmuth\"+\n" +
"\"+ Gary Weiss\"+\n" +
"\"- Sholom Weiss\"+\n" +
"\"- Thomas Wengerek\"+\n" +
"\"- Bradley L. Whitehall\"+\n" +
"\"- Alma Whitten\"+\n" +
"\"+ Robert Williamson\"+\n" +
"\"+ Janusz Wnek\"+\n" +
"\"+ Kenji Yamanishi\"+\n" +
"\"+ Takefumi Yamazaki\"+\n" +
"\"+ Holly Yanco\"+\n" +
"\"+ John M. Zelle\"+\n" +
"\"- Thomas Zeugmann\"+\n" +
"\"+ Jean-Daniel Zucker\"+\n" +
"\"+ Darko Zupanic\";";
```
HUFFMANN CODING IMPLEMENTATION
```
package huffman01;
import java.util.*;
import java.io.*;
public class Huffman01{
 public static void main(String[] args){

    Hashtable <Character,String>huffEncodeTable;
    //Insert here chosen Raw Data
    System.out.println("Raw Data");
    display48(rawData);

    int rawDataLen = rawData.length();

    System.out.println("nNumber raw data bits: "
                                + rawData.length() * 8);
    //Instantiate a Huffman encoder object
    HuffmanEncoder encoder = new HuffmanEncoder();
    huffEncodeTable = new Hashtable<Character,String>();
    ArrayList<Byte> binaryEncodedData = encoder.encode(
                                 rawData,huffEncodeTable);
     System.out.println("Number binary encoded data bits: "
                          + binaryEncodedData.size() * 8);
    System.out.println("Compression factor: "
      + (double)rawData.length()/binaryEncodedData.size());
    System.out.println(
            "nBinary Encoded Data in Hexadecimal Format");
    hexDisplay48(binaryEncodedData);
    System.out.println();
        HuffmanDecoder decoder = new HuffmanDecoder();
        String decodedData = decoder.decode(binaryEncodedData,
                                   huffEncodeTable,rawDataLen);
    System.out.println("nDecoded Data");
    display48(decodedData);
  }//end main
  static void display48(String data){
    for(int cnt = 0;cnt < data.length();cnt += 48){
      if((cnt + 48) < data.length()){
        System.out.println(data.substring(cnt,cnt+48));
      }else{
        System.out.println(data.substring(cnt));
      }//end else
    }//end for loop
  }//end display48
  static void hexDisplay48(
                      ArrayList<Byte> binaryEncodedData){
    int charCnt = 0;
    for(Byte element : binaryEncodedData){
      System.out.print(
               Integer.toHexString((int)element & 0X00FF));
      charCnt++;
      if(charCnt%24 == 0){
        System.out.println();//new line
        charCnt = 0;
      }//end if
    }//end for-each
  }//end hexDisplay48
  }//end class Huffman01
class HuffmanEncoder{
  String rawData;
  TreeSet <HuffTree>theTree = new TreeSet<HuffTree>();
  ArrayList <Byte>binaryEncodedData =
                                   new ArrayList<Byte>();
  Hashtable <Character,Integer>frequencyData =
                      new Hashtable<Character,Integer>();
  StringBuffer code = new StringBuffer();
  Hashtable <Character,String>huffEncodeTable;
  String stringEncodedData;
  Hashtable <String,Byte>encodingBitMap =
                            new Hashtable<String,Byte>();
  ArrayList<Byte> encode(
            String rawData,
              Hashtable <Character,String>huffEncodeTable){
    //Save the incoming parameters.
    this.rawData = rawData;
    this.huffEncodeTable = huffEncodeTable;
/*
    System.out.println("nRaw Data as Bits");
    displayRawDataAsBits();
*/
    createFreqData();


/*
    displayFreqData();
*/
    createLeaves();

    createHuffTree();

    createBitCodes(theTree.first());
/*
    System.out.println();
    displayBitCodes();
*/
    encodeToString();
    buildEncodingBitMap();
    encodeStringToBits();
        return binaryEncodedData;
  }//end encode method
    void displayRawDataAsBits(){
    for(int cnt = 0,charCnt = 0;cnt < rawData.length();
                                       cnt++,charCnt++){
      char theCharacter = rawData.charAt(cnt);
      String binaryString = Integer.toBinaryString(
                                          theCharacter);
      while(binaryString.length() < 8){
        binaryString = "0" + binaryString;
      }
      if(charCnt%6 == 0){
        charCnt = 0;
        System.out.println();
      }
      System.out.print(binaryString);
    }
    System.out.println();
  }
  void createFreqData(){
    for(int cnt = 0;cnt < rawData.length();cnt++){
      char key = rawData.charAt(cnt);
      if(frequencyData.containsKey(key)){
        int value = frequencyData.get(key);
        value += 1;
        frequencyData.put(key,value);
      }else{
        frequencyData.put(key,1);
      }//end else
    } }
  void displayFreqData(){
    System.out.println("nFrequency Data");
    Enumeration <Character>enumerator =
                                  frequencyData.keys();
    while(enumerator.hasMoreElements()){
      Character nextKey = enumerator.nextElement();
      System.out.println(
               nextKey + " " + frequencyData.get(nextKey));
    } }
 void createLeaves(){
    Enumeration <Character>enumerator =
                                  frequencyData.keys();
    while(enumerator.hasMoreElements()){
      Character nextKey = enumerator.nextElement();
      theTree.add(new HuffLeaf(
                     nextKey,frequencyData.get(nextKey)));
    }//end while
  }//end createLeaves
  class HuffLeaf extends HuffTree{
  private int value;

    public HuffLeaf(int value, int frequency){
      this.value = value;
      this.frequency = frequency;
    }
    public int getValue(){
      return value;
    }}
  void createHuffTree(){
    /*
    System.out.println("nnDisplay Original TreeSet");
    Iterator <HuffTree> originalIter = theTree.iterator();
    while(originalIter.hasNext()){
      System.out.println(
                     "nHuffNode, HuffLeaf, or HuffTree");
      displayHuffTree(originalIter.next(),0);
    }
*/
    while(theTree.size() > 1){
      HuffTree left = theTree.first();
      theTree.remove(left);
      HuffTree right = theTree.first();
      theTree.remove(right);
      HuffNode tempNode = new HuffNode(left.getFrequency()
                       + right.getFrequency(),left,right);
      theTree.add(tempNode);
/*
      System.out.println("nnDisplay Working TreeSet");
      Iterator <HuffTree> workingIter = theTree.iterator();
      while(workingIter.hasNext()){
        System.out.println(
                     "nHuffNode, HuffLeaf, or HuffTree");
        displayHuffTree(workingIter.next(),0);
      }//end while loop
*/
    }//end while
  }//end createHuffTree
  void displayHuffTree(HuffTree tree,int recurLevel){
    recurLevel++;
    if(tree instanceof HuffNode){
      HuffNode node = (HuffNode)tree;
      HuffTree left = node.getLeft();
      HuffTree right = node.getRight();
            System.out.print("  Left to " + recurLevel + " ");
      displayHuffTree(left,recurLevel);

      System.out.print("  Right to " + recurLevel + " ");
      displayHuffTree(right,recurLevel);
        }else{
      HuffLeaf leaf = (HuffLeaf)tree;
      System.out.println(
                        "  Leaf:" + (char)leaf.getValue());
    }
   System.out.print("  Back ");
 }
  class HuffNode extends HuffTree{
   private HuffTree left;
    private HuffTree right;
    public HuffNode(
               int frequency,HuffTree left,HuffTree right){
      this.frequency = frequency;
      this.left = left;
      this.right = right;
    }
  public HuffTree getLeft(){
      return left;
    }

    public HuffTree getRight(){
      return right;
    }}
  void createBitCodes(HuffTree tree){
    if(tree instanceof HuffNode){
        HuffNode node = (HuffNode)tree;
  HuffTree left = node.getLeft();
      HuffTree right = node.getRight();
      code.append("0");
      createBitCodes(left);
code.deleteCharAt(code.length() - 1);//Delete the 0.
      code.append("1");
      createBitCodes(right);
      code.deleteCharAt(code.length() - 1);
    }else{
      HuffLeaf leaf = (HuffLeaf)tree;
         huffEncodeTable.put((char)(
                         leaf.getValue()),code.toString());
    }}
  void displayBitCodes(){
    System.out.println(
          "nMessage Characters versus Huffman BitCodes");
    Enumeration <Character>enumerator =
                                huffEncodeTable.keys();
    while(enumerator.hasMoreElements()){
      Character nextKey = enumerator.nextElement();
      System.out.println(
            nextKey + " " + huffEncodeTable.get(nextKey));
    }}
  void encodeToString(){
    StringBuffer tempEncoding = new StringBuffer();
    for(int cnt = 0;cnt < rawData.length();cnt++){
      tempEncoding.append(huffEncodeTable.get(
                               rawData.charAt(cnt)));
    }
      stringEncodedData = tempEncoding.toString();
/*
    System.out.println("nString Encoded Data");
    display48(stringEncodedData);
*/
  }
  void  buildEncodingBitMap(){
  for(int cnt = 0; cnt <= 255;cnt++){
      StringBuffer workingBuf = new StringBuffer();
      if((cnt & 128) > 0){workingBuf.append("1");
        }else{workingBuf.append("0");};
      if((cnt & 64) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 32) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 16) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 8) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 4) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 2) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 1) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      encodingBitMap.put(workingBuf.toString(),
                                      (byte)(cnt));
    }}
  void encodeStringToBits(){
    int remainder = stringEncodedData.length()%8;
    for(int cnt = 0;cnt < (8 - remainder);cnt++){
      stringEncodedData += "0";
    }
    for(int cnt = 0;cnt < stringEncodedData.length();
                                         cnt += 8){
      String strBits  = stringEncodedData.substring(
                                        cnt,cnt+8);
      byte realBits = encodingBitMap.get(strBits);
      binaryEncodedData.add(realBits);
    } }
    void display48(String data){
    for(int cnt = 0;cnt < data.length();cnt += 48){
      if((cnt + 48) < data.length()){
        System.out.println(data.substring(cnt,cnt+48));
      }else{
        System.out.println(data.substring(cnt));
}}}}
class HuffmanDecoder{
  Hashtable <String,Character>huffDecodeTable =
                       new Hashtable<String,Character>();
  String stringDecodedData;
  String decodedData = "";
  Hashtable <Byte,String>decodingBitMap =
                          new Hashtable<Byte,String>();
  ArrayList <Byte>binaryEncodedData;
  Hashtable <Character,String>huffEncodeTable;
  //Used to eliminate the extraneous characters on the end.
  int rawDataLen;
  String decode(ArrayList <Byte>binaryEncodedData,
               Hashtable <Character,String>huffEncodeTable,
               int rawDataLen){
    this.binaryEncodedData = binaryEncodedData;
    this.huffEncodeTable = huffEncodeTable;
    this.rawDataLen = rawDataLen;
     buildDecodingBitMap();
     decodeToBitsAsString();
     buildHuffDecodingTable();
     decodeStringBitsToCharacters();
    return decodedData.substring(0,rawDataLen);
  }
   void buildDecodingBitMap(){
    for(int cnt = 0; cnt <= 255;cnt++){
      StringBuffer workingBuf = new StringBuffer();
      if((cnt & 128) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 64) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 32) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 16) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 8) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 4) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 2) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      if((cnt & 1) > 0){workingBuf.append("1");
        }else {workingBuf.append("0");};
      decodingBitMap.put((byte)(cnt),workingBuf.
                                             toString());
}}
  void decodeToBitsAsString(){
    StringBuffer workingBuf = new StringBuffer();
for(Byte element : binaryEncodedData){
      byte wholeByte = element;
      workingBuf.append(decodingBitMap.get(wholeByte));
    }
    stringDecodedData = workingBuf.toString();
  }
  void buildHuffDecodingTable(){
    Enumeration <Character>enumerator =
                                 huffEncodeTable.keys();
    while(enumerator.hasMoreElements()){
      Character nextKey = enumerator.nextElement();
      String nextString = huffEncodeTable.get(nextKey);
```