CHAPTER 1

DATA COMPRESSION
The technical terms used in this chapter are discussed as follows:

Data Compression: Any of many methods in information theory whereby data can be coded or recorded in order to take advantage of redundancy in the data.

Compression ratio: The ratio of length of original data to the compressed data.

Information Theory: The study of information by mathematical methods, where information can be considered as the extent to which a message conveys what was previously unknown, and so is new or surprising.

Reversible compression: The compression technique that allows decompression back to a form bit-for-bit identical with original data.

Non reversible compression: The compression technique, that produces only approximations to the original data, after decompression.

LZW Compression: Lempel-Ziv and Welch compression. A scheme for reversible compression of binary images in which codes are built adaptively to fit bit patterns encountered in data.

Run Length encoding: A means of compressing binary data by replacing strings of successive one’s or zeros by more compact binary code sequence.
1. Introduction.

We are entering the Information era, in which we act on the basis of learning how to use the laws of nature in order to get information available in sufficient amount and whenever needed. The tremendous progress in development of new information technologies is characteristic of our days. The information technologies comprise basically of the following three branches.

- Storing of information,
- Processing of information and
- Transmission of information.

These three types of information technologies have their specific features. Historically, they were developed as independent technologies. After the electronic computer was invented, it became the integration medium for these three branches of information technologies. Today, it is impossible to imagine communications without computers.

Information theory is a branch of communications science, introduced in 1948 by Claude Shannon. This theory has established a way of quantitative measuring of the information content of messages or strings of data. The information contained in a message written in a symbolic form is something different from the written text. The text is data and the meaning of the text is the information transmitted through this data. As the Shannon information...
theory is well known, we shall not go into the details of explaining the theory. However, the information theory of Shannon is astonishing in its generality, for it applies to all sources of information and media through which the information is sent. This theory is an instrument for finding optimal codes for transmission of information.

The information revolution made tremendous impact on everyday life. In the age of information explosion, the amount of data that is generated, stored and communicated has been so much that the Pundits of information handling have been forced to find new ways of storage and transmission of data.

1.1. Data Compression.

As the number of applications, users and number of databases increase, demand for storage space also increases. This would require additional disk drives or other storage media, leading to additional investment [32]. In the case of data communication, it is always desirable to have more amount of data to be transmitted in a minimum amount of time. We can increase the data transmission rate by upgrading the existing communication system or by installing special hardware and communication lines that can transmit more number of bits in a given time than the existing hardware. However, installing additional storage device and expanding the existing communication facilities, obviously lead to the establishment of new overheads and demand of more finance. And the expansion of additional resources in unlimited way
cannot be possible because of the limitations provided in the computer architecture. Thus, new ways of data management have to be explored. One of the ways of reducing the required space for storing the data is through data compression. It deals with representing data with efficient code, which can reduce the total number of bits used to represent the data [36]. That is the total volume of bits used to represent the same data is reduced, since number of bits in the compressed data is less than the number of bits in the original data.

 Transmitting compressed data through the existing hardware increases the transmission rate. The time needed to transmit a certain message is proportional to the number of bits in the message. Since the number of bits in the compressed data is always less than the original data, time taken for transmitting the compressed data will be less than the time taken for transmitting the original data. After reaching the destination, the transmitted data can be decompressed to get the original data. The transmission rate is the ratio of number of the amount of data transferred in unit amount of time. Since the number of bits in the compressed data is less, more data can be transferred in the given time, even though the actual data transfer expressed in bits per second remains the same. Hence effective transmission ratio is higher when compressed data is transmitted.
In the case of certain data base applications, by assigning certain codes to repeatedly occurring values, or by suppressing unused spaces in a record, one can reduce the storage space. Also we can omit the redundant data by assigning some relation to the data already stored. In this method, we may be able to reduce the total volume required to store the data. In order to apply this method of compression, one must know the logical relationship of the individual elements in the data. These are limited to only a few cases. This type of compression is named as logical compression, since this method is entirely data dependent and we cannot apply this technique to any data, which do not provide the relationship between data elements in it.

1.2 Physical and Logical Compression.

Physical compression can be viewed as the process of reducing the quantity of data by physically reducing the bits of storage. This compressed data can be kept in any medium and could be retrieved when needed. However, this data needs to be changed to its original format before using that data. In this method the data items are considered only as a stream of characters. The data items are not having any explicit relation between them.

A logically compressed data may be further compressed physically but the converse need not be true, because after physical compression one may not be able to separate out the individual data items and establish a relationship between them.
Theoretically lots of compression techniques are existing at present. One technique is based on encoding frequently occurring characters into short bit codes and infrequently occurring characters into longer bit codes. The volume of storage required to store the data encoded using this method, can be less than the volume required when data is encoded with same length of bit codes to all characters. In another method, replacing a repeating strings of characters by a character and a count telling how many times the same character is repeated, can reduce the total storage requirement [31].

It is obvious that by compressing the data, an overhead of processing time for compression and decompression exists. But this time factor is less in comparison with the time saved in transmitting the compressed data. The processing speed of CPU is far greater than the data transmission. The time taken for execution of a group of instructions is significantly less than the time taken to access and transfer data to a storage devices. Using the compressed data, the time taken for transmission is still reduced since the volume of data to be transferred is less. Thus the overall performance of the computer is enhanced by using most of the compression techniques, even though additional program instructions are executed for data compression and retrieval.
1.3. Reversible and Non reversible compression.

In most of the compression techniques, data is encoded in such a way that, decompression of the compressed data produces the original data itself [03]. Such type of compression, where the original data is identical to the data obtained after decompressing the compressed data is called as reversible compression. Certain compression techniques used for compressing graphic images produce only an approximation to the original data after decompression. A graphic image with 256 colors (gray scale values) would require at least 8 bits to represent one pixel on the screen. This image when displayed on a screen that is capable of producing only 16 colors will use only 4 bits per pixel. Hence this image data could be compressed to half of its original size, by using only 4 bits to represent one pixel. However, decompression of this compressed image would not produce an image that is identical to the original image. Such type of compression techniques that produce only an approximation to the original data after decompression is called as non reversible compression.

1.4 Compression Ratio.

Out of the given data the degree of data reduction obtained from the compression process is known as compression ratio. This ratio is a measure of the quantity of compressed data to the quantity of the original data.
i.e. Compression Ratio = \frac{\text{Length of original data string}}{\text{Length of compressed data string}}

Thus the higher compression ratio indicates that the data is compressed effectively and the size of compressed data is very less than the size of the original data. Higher compression ratio indicates that the compression technique is considered to be a more effective compression technique. The compression ratio for a particular technique will vary depending on the data to be compressed. The technique which gives maximum compression ratio for text files may not give the same value of compression ratio in the case of binary files [07].

For different kinds of data, different types of compression algorithms are available at present. Most of them are functionally the same, but their implementation techniques vary greatly. A few algorithms which produce higher compression ratio in the case of storing screen images [27] (using quad tree to store bitmap), would not be effective to produce significant results when applied to the data present in a data base file.

2. Compression Techniques.

In this chapter we study most popular and implementable compression techniques. We design and develop effective algorithms to implement these techniques of data
compression. These algorithms are translated to 'C' code and the software packages thus created in this process are used for compressing various types of data files. Our experiences in bringing out these packages on data compression are presented in this chapter.

2.1 Run Length Encoding.

This is one of the common and oldest method of data compression. In this method, if a character c is repeated n times, then the whole portion of these occurrences could be replaced by a symbolic representation [31]. This representation contains a special character denoting the compressed data followed by the character c itself and a count character having value n. The special character which represents compression will be useful when decompressing the compressed data to produce the original data. This method is explained in the following lines with an example.

Consider the string of text ABBBBBBBBCCCCCDD, in which, the character 'B' is repeated 8 times, the character 'C' is repeated 6 times. Assume that the character, say Ø, never appears as a part of the data. Hence we can use this character Ø as the special character prefix to represent compressed data stream. The first character in the stream A is not repeated. Hence it cannot be compressed, and it will be copied to output string as such. Since the next character B is repeated 8 times, this will be compressed and copied to the output string as ØBm, where m stands for the repeat
count and is represented in binary form in one byte. Similarly the character C which is repeated 6 times will be compressed as ⊘Cn, where n represents 6.

Thus the data after compression will be A⊘Bm⊘CnD, where m and n are respectively the representation for 8 and 6.

While decompressing the compressed data A⊘Bm⊘CnD of this example, the first character is A and hence it is copied to the output as a single character A. Since the next character happens to be ⊘, the character that follows it, B will be copied 8 times to the output string, since, the count m that follows B represent the number 8. In a similar way the character C will be copied 6 times to the output string, followed by the character D, resulting the output string as ABBBBBBBBCCCCCDD.
In this method, decompression process follows the following steps:

1. Read a character of compressed data string.

2. Compare the character with ®.

3. If the character is not ® transfer that character to the output string.

4. If the character is same as ® then the following two characters (two bytes), are read from the input string. The first character out of them is transferred to output string and the same character is repeated in the output string as many times as the number represented in the second byte.

In the previous example, the source string was 16 character long, and the compressed string was of length 8 characters. Thus the compression ratio is 16/8 = 2. If the characters B and C were repeated say 16 and 14 times, making the total string length as 32 characters, then applying the same method will again produce a compressed string of 8 characters only. Here in the compressed string, the only difference will be that the values represented by m and n will be 16 and 14 respectively instead of 8 and 6. Thus the compression ratio becomes 32/8 = 4. Hence the same compression technique produces different compression ratios for different sets of strings of data.
Since each compressed data group takes 3 characters, this technique will be effective, only when the characters are repeated more than 3 times. This method will produce better compression ratio, if the repeat count of characters are larger. In the above example, the length of compressed data remained same although the repeat count increased from 8 characters to 16 characters.

If the characters are represented by one byte, the maximum binary number that we can store in a character is 255. Since the repeat count which is also represented as a character can have a maximum value of 255, storage representation needs to be changed when a character is repeated more than 255 times. In this case, we use two consecutive bytes, one representing 255 characters and the other group representing the remaining number of characters. This method could be extended to any number of times for any large number of repetitions, using the consecutive bytes.

The best possible compression ratio in this method is $255/3 = 85$, when all the characters in the data are repeated maximum 255 times. And in the worst case (i.e. if no character is repeated more than 3 times), the compression ratio is 1.

Minor limitation in this method is on the selection of special character for representing the symbol of compressed characters, since all the characters in the character set
are expected to appear as a part of the data. In such cases, we will have to modify this technique in order to take care of such cases. One can select the least occurring character as the special symbolic character and assume its appearance as a character with repetition count. The repetition count could be 1 if it is not repeated. The only overhead in this case is that one character is represented by three characters, in the worst case when that character is present as part of data. However this is a rare case, and in general the total length of remaining characters will be considerably large so that this overhead will become negligible.

A large number of variations to this technique are available. One of them is the Null Suppression method used in IBM 3780 BISYNC transmission protocol [21]. In this method, the repeated null characters are replaced with a two character combination - one of them a special character representing the compression and the other a repeat count.

Since only null characters are replaced, the third character is not necessary. As in the general case, this technique also will give a large compression ratio, if the nulls are repeated more number of times.

2.2 Frequency Dependent Coding.

In this method, data is coded with codes of varying sizes instead of using fixed size codes to represent each character. If the code for the most frequent characters were
shorter than the code for remaining characters, the total number of bits used to store the data becomes less. For generating such type of codes, the probability of occurrence of each character should be known in advance.

Using the method of Frequency Dependent coding, the data that is to be compressed is scanned once and a frequency table is created for the occurrences of each character in the data. The characters are arranged according to the order of their frequency in the data and are assigned codes of varying length, with shorter codes for characters which are more frequent and longer codes for the remaining characters. Thus the average code length in terms of number of bits used will be less than the average code length when fixed size code is used. The total number of bits of the compressed data can be found out by summing the product of frequency with code length of each character.

Morse code is an example of such frequency dependent code in which the character e is assigned the shortest code of a single dot, and the other less frequent characters, a sequence of dashes and dots. D.A. Huffman developed a method of creating frequency dependent code. The Huffman code may be generated by arranging the characters of different frequencies in the form of a tree structure. The details of which are explained as follows.

While scanning the given data the frequency of each character is found and a frequency table is created. The frequency of each character corresponds to the probability
of occurrence of each of them. Two smallest probabilities from this table are found and a new probability equal to the sum of two probabilities is added to the table. The two characters with the smallest probabilities are kept as the child nodes of a tree, whose parent node is assumed as a dummy character with the probability equal to the sum of these two probabilities. After creating this subtree, include the newly created dummy character and its probability in the frequency table. Exclude these child nodes from the frequency table and construct a new subtree in the same manner by finding the two lowest probabilities from the updated frequency table. Continue this process till there are only two elements in the frequency table and the sum of their probability is 1. These two characters can then be considered as the child node of the root node with probability 1. The codes to each characters is now assigned as the path to each of them from this root node.

Assume the characters C1, C2, C3 and C4 has the probability say, 0.1, 0.15, 0.3 and 0.45 respectively. The tree may be created as follows.

```
   (0.45)  
   |      |     
C4    C3    
     |      |     
   (0.3)  
   |      |     
C3    C2    
     |      |     
   (0.15) 
   |      |     
C2    C1    
   |      |     
(0.1)  
```

26
The node with probability as 1 is the root node of this character tree. The codes corresponding to each character are derived from the path connecting that character from the root node. The path may be represented as a 0, if it is towards left, and is represented as 1, if it is towards right of the parent node. In this example, character C4 is reached from the root node by traversing only once through the right branch of a subtree. Hence the code for C4 is 1. In the same manner, the character C3 could be reached, from traversing through the left branch to reach the dummy node and from there traversing through the right branch of the tree. Hence the code corresponding to C3 is 01. Following the same method, the codes corresponding to C1 is 000, C2 is 001. The average number of bits used for representing each character using this method is the sum of products of probability with the code length of each character. In this example it is $3 \times 0.1 + 3 \times 0.15 + 2 \times 0.3 + 1 \times 0.45 = 1.8$. If we were using fixed size code, in order to represent 4 characters, the average code size would be 2.

While decompressing the data that has been encoded using Huffman code, compressed data is scanned bit by bit. If the bit read represents a character, that character may be copied to the output. Else read one more bit and find out whether these two bits represents a character. Repeat this process till the number of bits in the compressed data is exhausted. The original data can be extracted character by character as the compressed data is scanned bit by bit. In this case one need not have to wait for the special
character or the character count to start the decompression. Each character can be extracted, as soon as the character code is encountered.

It is necessary to note that this method required perfection in data handling, as change in a bit could produce disastrous result. For example, if one bit of the compressed data gets changed, the decompression may give a totally garbled result. Also if the decompressing routine looses the track of one bit, the stream of bits that follows, may be interpreted in an entirely different way, since the codes are of varying length.

The Huffman code created for one set of data need not be efficient for another set of data. If the Huffman code is created for alphabets of English, using normal text, the character e may have the smallest code, since the character e has the maximum frequency. If the same code is used for a program written in any of the programming languages, the expected compression might not be obtained. Thus it is always advisable to scan the data and create the most appropriate code which produces the maximum compression.

If P is the probability of appearance of a particular character in the data, the number of bits used to code it in this method will be the nearest integer value greater than \(-\log_2 P\). If the probability of occurrence of a character is very less, it may so happen that the code of that character is longer than 8 bits [21].

28
2.3 Bit Mapping.

Bit mapping [21] is used to replace the occurrence of a particular character with bit maps. This technique is based on the usage of binary digits 1 and 0 to represent the presence or absence of certain characters in the data. A character say, Null character may occur quite frequently in the data, but is not in the repeated form, so that the earlier algorithm fails to give a good compression ratio. In this case by keeping a bit mask of 0 to represent the absence and 1 to represent the presence of the Null character may reduce the size of data after compression.

During compression, the first byte in the output is used as a bit mask representing the presence or absence of Null character. If the first byte of the input string is a Null character, keep the first bit in the bit mask as 1. Otherwise, keep the first bit in the bitmask as 0 and the byte following the bitmask as the input character that is read. Similarly if the second byte is null keep the second bit of the bitmask as 1. When the bitmask is full, that is when 8 characters are read, another byte is copied to output stream. This byte will be used as the bit map for another 8 bytes of input data. The procedure is continued till the end of the input string. This procedure could be explained in the following lines.
Consider the following string of 8 characters,

\[
\begin{array}{c}
A \ NULL \ B \ NULL \ NULL \ C \ NULL \ NULL \\
\end{array}
\]

We use the above technique to the sample data and produce the compressed data as follows:

\[
\begin{array}{c}
\text{Mask! A ! B ! C !} \\
\end{array}
\]

Where Mask is \[0!1!0!1!1!0!1!1;\]

In the bit mask, first bit is 0, since the first character is not NULL, second bit is 1 since NULL follows the character A. Similarly third and sixth bit in the bit mask is 0's and remaining bits are 1's.

In this example, the total length of the compressed string is 4 bytes. Out of the original string of 8 bytes, giving a compression ratio of 2. On applying the Null suppression technique to this data would not have produced any compression at all. If \( P \) is the percentage of NULL characters in the string, then the compression ratio could be expressed as \( \frac{800}{(900 - 8P)} \). Using this formula it is clear that, the maximum compression ratio will be \( \beta \), when the percentage of NULL is 100, and minimum compression ratio is 0.888 when there is no NULL character in the input string. This technique may increase the size of the resulting string, if the frequency of NULL is less than 12.5% of the total length of input string.
A variation of this technique known as Half Byte Packing is used to compress data that is to be stored as digits. If we are using EBCDIC or 8 bit ASCII code to represent these digits, the first four bits will be same for all digits. Hence we need to store only the lower nibble to represent the corresponding character, making a single byte to store two digits. This technique will be useful only in a few cases, where a group of bits in each consecutive character of the data is same.

The most recent LZW compression (named after Lempel, Ziv and Welch) used for compressing binary data, gives very high compression ratio. In this method the system builds up a table of tokens as it scans the data for compression. It identifies a group of bytes with a token and makes an entry in the table of tokens. When the same group of bytes is encountered again, it will be replaced by the token from the table of tokens. Thus, a chunk of data would be replaced by a token, there by reducing the total data size.

3. Preamble to our software DATACOMP.

These are some of the general techniques used for data compression. The commercial packages that are available in the market do not provide much documentation and we are not aware of the precise algorithms that are implemented by those packages for handling different kinds of data. Apart from their price tag, these commercial packages do not provide any clue about the organisation of compressed data.
We have studied a large number of compression techniques and analyzed their effectiveness on different kinds of data. In this chapter we have designed developed and implemented some useful techniques for data compression. All the techniques used in this chapter are modified versions of existing techniques in the literature and they are implemented using C language more efficiently for handling data compression.

4. **DATACOMP implementation.**

The package that we have developed for data compression, first analyses the data to select the best method of compression which gives maximum compression ratio and then compresses the data using the selected technique. In order to make the decompression faster, a code indicating the type of compression technique is added to the data. Though the time taken for compression is more, time taken for decompression is less. Few of the techniques used in the package is explained in the following lines.

4.1. **Method 1 : Text compression under DATACOMP.**

One technique that we have developed is for compressing ASCII text. This technique uses the method of Frequency Dependent Encoding. If the size of the input data is smaller, and the number of different characters is less, the data is encoded with already created Frequency Dependent Code which is giving maximum compression [16]. Otherwise the data is scanned to find the frequency of occurrence of each
of the character and the code is generated depending on the probability of occurrence. The data is encoded with the new codes that are generated, which give maximum compression ratio. The code table is appended at the end of the compressed data. The first byte of the compressed data tells whether the encoding is done with already existing code, or newly created code. In the case of compression using the newly created code, the four bytes following the first character is used to store the offset of the beginning of code table that is used to encode the data.

4.1.1. Compression Algorithm 1. (For Compression of Text)

1.0 Initialisation.

1.1 Initialise the code table. This table contains 255 items each of which contains a byte representing the code length and a pointer to string of bytes, the code with initial values as 0.

1.2 Initialise the frequency table. This table initially contains 255 integers (long), representing the frequency, with initial value 0 and each with two pointers, namely left and right pointers having NULL values.

2.0 Construct the frequency table for the given data.

2.1 Read a character from the data file and increment the counter representing the size of data.
2.2 Increment frequency corresponding to the ASCII code of the character that is read in the frequency table.

2.3 Repeat the above two steps till the end of file is reached.

3.0 Convert the frequency table to a Frequency tree.

3.1 Find two elements with lowest frequency which is greater than zero from the frequency table.

3.2 Add a new element to the frequency table, with the frequency equals the sum of these two frequencies. Make the two pointers of this new element to store the index corresponding to the positions in the frequency table of these two elements with lowest frequency.

3.3 Replace the frequency of the two elements obtained from step 3.1 with a negative value whose modulus is equal to the index (position in the table) of newly added element in the frequency table. This number is used to point to the parent of the node in the tree.

3.4 Repeat the steps 3.1 to 3.3 till the frequency of the newly added element equal to the size of data obtained from step 2.1
4.0 Construct the code for each character.

4.1 Find the parent of the element (element whose index is the modulus of the field which was initially containing frequency) in the frequency table corresponding to the given character. If the frequency is zero, skip steps from 4.2 through 4.4.

4.2 If the left pointer of this parent node is equal to the position (index) of the previous node, add a bit with value 0 or otherwise add a bit with value 1 to the code entry, and increment code size of the item corresponding to the character in the code table.

4.3 Consider this parent node as a child and find its parent node.

4.4 Repeat steps 4.2 through 4.3 till the frequency field of the element is non negative.

4.5 Take the next element in the frequency table and repeat steps 4.1 through 4.4 till the codes corresponding to all the 256 characters are known.

5.0 Encode the data.

5.1 Reset the data pointer to the beginning of data. Copy a byte to the output stream, representing the encoding technique. Assign a new byte to the output stream and keep the counter for available bits in the output stream, say bits_avail as 8, and current bit position, say curr_pos as 1.
5.2 Read a character from the input data stream, and find the code and its length for this character from the code table.

5.3 If the code length is less than bits_avail, copy the bits of the code to the output stream starting from the curr_pos. Increment the curr_pos and decrement the bits_avail by the size of the code.

5.4 If the code length is greater than or equal to bits_avail, then

5.4.1 Fill the available position in the output stream with bits_avail number of bits of the code. Exclude these bits of the code and decrement the code length by bits_avail.

5.4.2 Assign one more byte to the output stream, initialise bits avail with 8 and curr_pos with 0.

5.4.3 Repeat executing previous steps from the step 5.3 with new values of code length and bits_avail.

5.5 Repeat the steps from 5.2 to 5.4 till all the characters from the input stream is exhausted.
6.0 Append the code used for encoding, in the case of large volume of data.

6.1 Find the size of output bytes already created. This will be used to note down the offset of beginning of code.

6.2 Copy the code length of the character and number of bytes used as code to the output stream. If code length is zero, no extra bytes will be copied.

6.3 Repeat the previous step for all characters in the code table.

6.4 Finally copy the offset of the beginning of code to the output stream and close it.

4.1.2. Decompression Algorithm 1. (For Decompression of Text)

1.0 Initialise the code table.

1.1 Get the offset of the code in the compressed data and keep the data pointer to this location.

1.2 Read a byte from the current position. This byte represents the length of code, say code_len. If code_len is not zero, let code_bytes be the smallest integer greater than code_len/8. Read code_bytes number of bytes from the current position, which represent the code corresponding to the character.
1.3 Repeat step 1.2 till the code for all the 256 characters are known.

1.4 Add another field to each element of the code table representing the character corresponding to the code. Sort the code table in the ascending order of code size.

2.0 Construct the original data.

2.1 Place the data pointer to the beginning of compressed data.

2.2 Read a byte from the input stream, and extract the first bit from the current byte and assume it as the current code.

2.3 Find out whether a character corresponding to the current code is existing in the code table. If it exists copy the character to the output stream and discard the current code.

2.4 If all the bits of current byte used and no character corresponding to the current code is found out, read the next byte of the input as the current data byte. Extract next bit from the current byte, and append it to the current code. If there is no more bytes available in the compressed data, discard the current code and close the output stream since, these bits are extra bits padded to the compressed bits to make a byte.

2.5 Repeat steps 2.4 to 2.5 each time taking the next bit as the current code, till all the compressed data bytes are exhausted.
4.2. Method 2: Binary data compression under DATACOMP.

Another technique is based on the Run length encoding. In this method, instead of using three bytes to represent repeated characters, only two bytes are used. The first byte represents both the special character representing compression as well as the repeat count. If the most significant bit of a character is set, it is assumed as the special character representing compression. The remaining seven bits of this byte will give the repeat count of the character following it. If the first bit is not set, then this byte represents a count of uncompressed data characters following it.

4.2.1. Compression Algorithm 2. (For Repeated data)

1. Initialise two counters namely Rept_count and Ncn_Rept_count which represent the count of repeating characters and the count of non-repeating characters respectively with 0.

2. Open the input stream of data to be compressed and read a character, consider this as prev_char.

3. If the end of file is not reached, read another character say curr_char.

3.1. If prev_char = curr_char, then

3.1.1. If Non_Rept_count > 0, then copy this number at the byte reserved for it in the output stream. Reinitialise Non_Rept_count with 0.
3.1.2 If Rept_count = 127 then write down this number to the output stream followed with the character prev_char.

3.1.3 Increment the Rept_count.

3.2. If prev_char is not equal to curr_char, then

3.2.1 If Rept_count \( > 0 \), then add 128 to this count and write down this number to the output stream followed with prev_char. Replace Rept_count with zero.

3.2.2 If Non_Rept_count = 0, then write down a byte reserved for the Non_Rept_Count.

3.2.3 If Non_Rept_count = 127, then copy the number 127 at the byte reserved for it in the output stream. Replace Non_Rept_count with 0.

3.1.3 Increment the Non_Rept_count. Add the prev_char to the output stream and replace prev_char with curr_char.

4. Close the input data stream.

4.1 If Non_Rept_count > 0, then copy Non_Rept_count to the byte reserved for it and add curr_char to the output stream.
4.2 If Rept_count > 0, then add 128 to this count and write down this number to the output stream followed with curr_char.

5. Close the output stream.

4.2.2. Decompression Algorithm 2. (For Repeated data)

1. Open the input stream of compressed data and read a byte, say Count.

2. If the Most Significant Bit of this Count is set, then

   2.1 Read the next character from the input data.

   2.2 Copy this character to the output stream Count-127 times.

3. If the Count is < 128 then

   3.1 Read the next character from the input stream and copy it to the output stream without any change.

   3.2 Repeat step 3.1 Count number of times. If end of file is reached in between, the input data may be corrupted.

4. If end of file is not reached, read the next byte to Count and repeat step 2 to 3 with this new value of Count.

5. Close the input and output stream.
4.3. Decision Support System under DATACOMP.

Having implemented data compression techniques for handling different types of given data files, we have discovered that there is a need for a Decision Support System to help in analysing the given data form and identifying the suitable compression technique and finally help in compressing the data, using the best possible compression techniques. For example, Frequency dependent coding technique to compress a binary file (like executable files under DOS, or any other binary data files) would not give good compression ratio. The Run length encoding technique to store screen images data (of graphics applications) could give a higher compression ratio than other techniques. The LZW algorithm gives higher compression ratio for compressing binary data where the other two techniques gives low compression ratio.

Using these objects (software tools for different methods), we have built up a decision support system. This system scans the input data file apply different methods of data compression and finally selects the best possible compression techniques applicable on this data file. Needless to say that the given data file is compressed using the selected techniques and outputs the compressed data file.