CHAPTER 2

LOSSLESS DATA COMPRESSION ALGORITHMS

Data compression is a fairly mature field of work. There are many different types of compression algorithms which are designed to compress different types of data. In spite of intensive research no compression algorithm has yet been discovered that consistently attain the predictions of lower bound of data compression [78] over wide classes of text files. One approach in the lossless text compression area is to find better algorithms to explore the redundancy of the context and achieve a better compression ratio with a good time complexity. This chapter focuses on lossless text compression algorithms wherein various methods like Statistical compression techniques, Dictionary-based compression techniques and Predictive compression techniques are presented. At the end, a comparative study of Statistical and Dictionary- based compression techniques is presented.

2.1 STATISTICAL COMPRESSION TECHNIQUES

In the statistical coding, every symbol is encoded according to the frequency in which it occurs. Shorter codes are assigned to the symbols which occur frequently and longer codes are assigned to the symbols which have less frequency. The statistical methods are rarely used as independent compression methods. They are usually used as the coding methods in two stage lossless data compression (modeling and coding).

Processing a data stream is done in its entirety by the sequential statistical data compression algorithms. In this method, constructing and refining a model based on the information that has been processed is done without the need of accessing symbols any further upstream than the first un-encoded one.
A compressor built in this way has the following advantages:

- Since all the information that the compressor uses is available to the decoder, there is no need to output extra data for the decoder in order to enable the decoder to find out how the data are encoded.
- Files of any size can be processed under this coding and the size of the file that has to be compressed need not be known before the process is started. The size of the file may be arbitrarily large, as one might imagine to be the case for a data channel between two computers on a network (the compression serving to improve bandwidth).
- The coding, more significantly, simplifies the process conceptually. Symbols from the information source are either encoded or not encoded. There is never any correlation between the output and as yet unencoded input symbols.

The Figure 2.1 given in the illustration will demonstrate that the sequential statistical data compressor is a result of the sequence of two data stream filters. As and when new input symbols are received, the source model module always tries to adaptively model the information source. With the help of the generated model the second module would encode the data (symbols) efficiently, which was received from the previous module. Similarly in the same way the model can be constructed in the process of decompression. Since new symbols are decoded it makes the model an updated one.

![Figure 2.1: Sequential Statistical Data Compressor](image)

25
In this section, the following statistical compression algorithms are discussed.

- Run Length Encoding technique
- Shannon-Fano coding
- Static Huffman coding
- Adaptive Huffman coding and
- Arithmetic coding

2.1.1 RUN LENGTH ENCODING TECHNIQUE (RLE)

One of the simplest compression techniques known as the Run-Length Encoding (RLE) [34] is created especially for data with strings of repeated symbols (the length of the string is called a run). This algorithm is simple, fast and effective for a source that contains many long runs. The main idea behind this is to encode repeated symbols as a pair (l,s) where ‘l’ is the length of the string and ‘s’ is the symbol. For example, the string ‘abbaaaabaabbbaa’ of length 16 bytes (characters) is represented as 7 integers plus 7 characters, which can be easily encoded on 14 bytes (as for example ‘1a2b5a1b2a3b2a’).

This algorithm is very effective if the source text contains many runs of consecutive symbols. The symbols may be characters in a text file, 0’s and 1’s in a binary file or colour pixels in an image or even component blocks of larger sound files. Simple run-length algorithms are widely used in practice. HDC (Hardware Data Compression), an algorithm which is used by tape drives connected to IBM computer systems and another algorithm used in IBM SNA (System Network Architecture) standard which is similar to HDC, are in vogue even today for data compression.
The biggest problem with RLE is that in the worst case the size of output data can be two times more than the size of input data. To eliminate this problem, each pair (the lengths and the strings separately) can be later encoded with an algorithm like Huffman coding.

For compression of gray scale images, Run Length Encoding can also be used. Each run of pixels of the same intensity (gray level) is encoded as a pair (run length, pixel value). The run length manifests itself either as emission as one byte, allowing for runs of up to 255 pixels, or as encoding by a variable-length code. The pixel value is encoded in a short fixed-length code and its length depends upon the number of gray levels (typically between 4 and 8 bits).

RLE with its modifications are widely used, especially in well-known graphics files like: BMP images, PCX images, and TIFF images. RLE is often combined with other compression algorithms. RLE is used to reduce a number of zeros after Move-to-Front (MTF) encoding in some BWT-based (Burrows–Wheeler Transform) algorithms (for example, in [28] and after quantization in JPEG [61] image compression.)

### 2.1.2 SHANNON-FANO CODING

Shannon–Fano algorithm was simultaneously developed by Claude Shannon (Bell laboratories) and R.M. Fano (MIT)[27,77]. It is used to encode messages depending upon their probabilities. It allots less number of bits for highly probable messages and more number of bits for rarely occurring messages.
The algorithm is given below in Figure 2.2.

1. Construct a frequency or probability table for the symbols listed.
2. Arrange the table, placing at the top the symbol that figures most frequently.
3. Bifurcate the tables into two halves, keeping as close as possible the total frequency count of the upper half and the total frequency count of the bottom half.
4. Organise the divided tables in such a way assigning the upper half of the list a binary digit ‘0’ and the lower half a ‘1’.
5. Apply repeatedly the steps 3 and 4 to each of the two halves, subdividing groups and adding bits to the codes until each symbol has become a corresponding leaf on the tree.

**Figure 2.2: Shannon-Fano Encoding Algorithm**

**Example 2.1**

Consider the sorted frequency table

<table>
<thead>
<tr>
<th>Symbol</th>
<th>U</th>
<th>V</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>14</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

Bifurcate the table into two halves so that the sum of the frequencies of the upper and bottom halves are as close as possible.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>U</th>
<th>V</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>14</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>sum(s)</td>
<td>(21)</td>
<td></td>
<td>(16)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Assign the upper half of the list a binary digit ‘0’ and the lower half a ‘1’.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>U</th>
<th>V</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>14</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>sum(s)</td>
<td>(21)</td>
<td>(16)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Codewords</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Repeat the above recursively to each group.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>U</th>
<th>V,</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>14</td>
<td>7,</td>
<td>6</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>sum(s)</td>
<td>(14)</td>
<td>(7),</td>
<td>(6)</td>
<td>(10)</td>
<td></td>
</tr>
<tr>
<td>Codewords</td>
<td>00</td>
<td>01,</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

Shannon-Fano Code is

<table>
<thead>
<tr>
<th>U</th>
<th>V</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>01</td>
<td>10</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>

Generally, Shannon-Fano coding does not guarantee that an optimal code is generated. Shannon – Fano coding is more efficient when the probabilities are closer to inverses of powers of 2. Shannon-Fano coding is used in the Implode compression method for zip files.
2.1.3 STATIC HUFFMAN CODING

The Huffman coding algorithm [46] is named after its inventor, David Huffman, who developed this algorithm as a student in a class on information theory at MIT in 1950. It is a more successful method used for text compression. The Huffman coding algorithm is proven to be helpful in decreasing the overall length of the data by assigning shorter codewords to the more frequently occurring symbols, employing a strategy of replacing fixed length codes (such as ASCII) by variable length codes. The codes generated using this technique or procedure is called Huffman codes. These codes are prefix codes and are optimum for a given model (set of probabilities). The Huffman procedure is based on two observations regarding optimum prefix codes.

- In an optimum code, symbols that occur more frequently (have a higher probability of occurrence) will have shorter codewords than symbols that occur less frequently.
- In an optimum code, the two symbols that occur least frequently will have the same length.

The Huffman procedure is obtained by adding a simple requirement to these two observations. This requirement is that the codewords corresponding to the two lowest probability symbols differ only in the last bit.

Huffman algorithm is not very different from Shannon-Fano algorithm. Both the algorithms employ a variable bit probabilistic coding method. The two algorithms significantly differ in the manner in which the binary tree is built. Huffman uses bottom-up approach and Shannon-Fano uses Top-down approach.
Huffman Compression Algorithm

The Huffman compression algorithm comprises the following three steps: counting the character frequencies, construction of the prefix code and encoding of the text. In this process the later steps use the information processed by the earlier steps. In the first step the number of occurrences of each character in the original text is counted. If fixed statistics on the alphabet is used, this step can be avoided. In this case, however, the method is optimal according to the statistics, but not necessarily for the specific text.

The second step of the algorithm builds the tree of a prefix code, called a Huffman tree.

The procedure for building this tree is given in Figure 2.3.

1. Start with a list of free nodes, where each node corresponds to a symbol in the alphabet.
2. Select two free nodes with the lowest weight from the list.
3. Create a parent node for these two nodes selected and the weight is equal to the weight of the sum of two child nodes.
4. Remove the two child nodes from the list and the parent node is added to the list of free nodes.
5. Repeat the process starting from step-2 until only a single tree remains.

Figure 2.3: Procedure for Building Huffman Tree
After building the Huffman tree, the algorithm creates a prefix code for each symbol from the alphabet simply by traversing the binary tree from the root to the node, which corresponds to the symbol. It assigns 0 for a left branch and 1 for a right branch.

The unique prefix property is the special feature of the codes generated by this algorithm. Huffman codes which arrive in a stream can unambiguously be decoded since no code is a prefix to another code. The symbol with the highest probability has been assigned the fewest bits, and the symbol with the lowest probability, has been assigned the most bits.

In fact, D.A. Huffman proved in 1952 that his coding method cannot be improved by any other scheme that uses an integral number of bits to code characters. Both Huffman and Shannon-Fano use roughly similar time to compress/expand. But Huffman is preferred to the Shannon-Fano algorithm. It is why Huffman finds its way in many compression tools, JPEG algorithm and even FAX machines.

The algorithm presented above is called a semi-adaptive or semi-static Huffman coding as it requires knowledge of frequencies for each symbol from alphabet. Along with the compressed output, the Huffman tree with the Huffman codes for symbols or just the frequencies of symbols which are used to create the Huffman tree must be stored. This information is needed during the decoding process and it is placed in the header of the compressed file.
Example 2.2

Consider the string **JACK BEATS JILL**.

For convenience two blank spaces are ignored. Encoding the message using Static Huffman algorithm can be done as follows:

The frequency of each symbol is

<table>
<thead>
<tr>
<th>Symbol</th>
<th>J</th>
<th>A</th>
<th>C</th>
<th>K</th>
<th>B</th>
<th>E</th>
<th>T</th>
<th>S</th>
<th>I</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

The above list is sorted by frequency of occurrence

<table>
<thead>
<tr>
<th>Symbol</th>
<th>J</th>
<th>A</th>
<th>L</th>
<th>C</th>
<th>K</th>
<th>B</th>
<th>E</th>
<th>T</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Here, the source consists of symbols from an alphabet (J,A,L,C,K,B,E,T,S,I) with the frequency (2,2,2,1,1,1,1,1,1). Here, the goal is to assign a variable length prefix code to the symbols of an alphabet.

As shown in the Figure 2.4, the two symbols S and I are with the least frequencies and they are combined to form a new combined item SI with a frequency 2. The combined item SI is then inserted in the first position in order to maintain the sorted order of the list. It is important to note that SI with a frequency 2 can be placed in more than one possible ways. For example, it may be placed after or before J, A and L. Here, the newly combined item is always placed to the highest possible position in order to avoid getting combined soon again. So, SI is placed before J. Continuing in this fashion, the Huffman tree is obtained as shown in the Figure 2.4.
Figure 2.4: Construction of Huffman Tree
Once when the binary tree is built, it is easy to assign a 0 to the left branch and a 1 to the right branch for each internal node as shown in the Figure 2.5.

![Huffman Tree Diagram](image)

**Figure 2.5: A Huffman Tree**

The Huffman codes for the symbols of an alphabet are done as follows. For example, the collection of 0’s and 1’s from the root to leaf for the symbol S is first a left branch 0, then again left branch 0, then right branch 1 and finally left branch 0. So the codeword for S is 0010.

Traversing in this way for all the leaves, the prefix code derived is as follows:

<table>
<thead>
<tr>
<th>J</th>
<th>A</th>
<th>L</th>
<th>C</th>
<th>K</th>
<th>B</th>
<th>E</th>
<th>T</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>010</td>
<td>011</td>
<td>100</td>
<td>101</td>
<td>110</td>
<td>111</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
</tr>
</tbody>
</table>

So, the message JACK BEATS JILL can then be encoded as

```
010 011 101 110 111 0000 011 0001 0010 010 0011 100 100
```
It is clear that symbol J has a higher frequency than C and so the codeword length for J should be no longer than C. The longest codewords should be assigned to the more rare symbols. Here the more rare symbols are the last two symbols in our sorted list. So the codeword for S is 0010 and for I it is 0011. This satisfies the requirement that the codewords corresponding to the two lowest probability symbols differ only in the last bit.

**Canonical and Minimum-Variance Huffman Coding**

The following two rules are generally followed as standard practice during the derivation of a Huffman tree.

- In order to keep the list sorted, a newly created item is placed at the highest possible position in the alphabet list.
- When combining two items, the one higher up on the list is assigned 0 and the one lower down is assigned 1.

The Huffman code derived from a process that follows these rules is called a canonical and minimum-variance code. The code is regarded as standard and the length difference among the codewords is kept to the minimum.

**Huffman Decompression Algorithm**

The decompression algorithm involves the operations where the codeword for a symbol is obtained by walking down from the root of the Huffman tree to the leaf for each symbol.
The Huffman decoding algorithm is outlined below in Figure 2.6.

- Read the coded message bit by bit.
- Starting from the root node, traverse one edge down the tree to a child according to the bit value. If the current bit read is 0 move to the left child, otherwise, to the right child.
- Repeat this process until a leaf is reached. If a leaf is reached, decode one character and restart the traversal from the root.
- Repeat this read-and-move procedure until the end of the message is encountered.

**Figure 2.6: Huffman Decompression Algorithm**

**Example 2.3**

Decoding the sequence 000000010011 using the Huffman tree generated is shown below.

Figure 2.7 shows the 12 steps of decoding the symbols E, T and I. The decoder reads the 0s or 1s bit by bit. For example in step 1, starting from the root of the Huffman tree we traversed the left branch one down to the left child since a bit 0 is read. In step 2, again we moved along the left branch again to the left child since a bit 0 is read and so on. When a leaf is reached, the symbol at the leaf (i.e. E) is given as output. This process is again repeated from the root and continues until another leaf is reached. The decoding process ends when EOF is reached for the entire string.
Figure 2.7: Decoding the Sequence 000000010011
Figure 2.7: Decoding the Sequence 000000100111 (contd.)
The Huffman coding is rarely used as an independent compression method. It is usually used as the coding method in the last stage of lossless data compression. Huffman compression is used in connection with, for example, LZSS algorithm (used in gzip, PKZip, ARJ, and LHArc), BWT-based algorithms, JPEG [94] and MPEG compression, Run-Length Encoding and Move-To-Front coding. Most of these algorithms are described in this dissertation.

2.1.4 ADAPTIVE HUFFMAN CODING

The probability distribution of the input set is required to generate Huffman codes. The basic Huffman algorithm is handicapped by the drawback that such a probability distribution of the input set is often not available. The probability distribution table is sent along with the compressed data only because of this reason. Although this overhead is not very significant even for small files a demand arises that instead of order-0 model, an order-1 model be used for better compression scheme. The overhead associated with sending the probability distribution table gets too high when an order-1 model is used. In addition to that, it is found to be unsuitable to cases when there is a change in the probabilities of the input symbols. So, an adaptive mechanism called the Adaptive Huffman coding is used with an explicit objective of achieving better compression without incurring the high overheads (of sending probability distribution table). The compression ratio is improved by Adaptive Huffman coding algorithms through application to the model the statistics which has as its basis the source content seen from the immediate past. A dynamic adjustment of an alphabet and its frequency table is made after reading each symbol during the process of compression or decompression. The Adaptive Huffman coding technique was developed based on Huffman coding first by Newton Faller [26] and by Robert G. Gallager[33] and then improved by Donald Knuth [50] and Jefferey S. Vitter [92,93].
In this method, a different approach called sibling property is introduced to build a Huffman tree. Dynamically changing Huffman code trees, whose leaves are representative of the characters seen so far, are maintained by both the sender and receiver. Initially the tree contains only the 0-node, a special node representing messages that have yet to be seen. Here, the Huffman tree includes a counter for each symbol and the counter is updated every time when a corresponding input symbol is coded. Huffman tree under construction is still a Huffman tree if it is ensured by checking whether the sibling property is retained. If the sibling property is violated, the tree has to be restructured to ensure this property. Usually this algorithm generates codes that are more effective than static Huffman coding. Storing Huffman tree along with the Huffman codes for symbols with the Huffman tree is not needed here. It is superior to Static Huffman coding in two aspects: it requires only one pass through the input and it adds little or no overhead to the output. But this algorithm has to rebuild the entire Huffman tree after encoding each symbol which becomes slower than the static Huffman coding.

The Adaptive Huffman encoding and decoding process can be outlined in Figure 2.8 and 2.9 respectively.

```
Initialize_model();
while ((ch = getc (input)) != eof)
    encode (ch, output);
    update_model (ch);
end while
```

**Figure 2.8: Adaptive Huffman Coding – Encoding Process**
Initialize_model();
while ((ch = decode (input)) != eof)
    putc (ch, output);
    update_model (ch);
end while

Figure 2.9: Adaptive Huffman Coding – Decoding Process

The most noteworthy point of this is that both encoder and decoder use exactly the same initialization and update_model routines. For implementation of the initialization routine several alternatives are available. One of the options is to use only two codes, namely EOF and ESCAPE at the beginning of the execution. EOF signifies the end of the input string. ESCAPE code indicates that the bit-string that follows is the first occurrence of a character that is not yet present in the Huffman tree. This code is necessitated since, initially, no code corresponding to any given character is present there. When a character comes across in the input string for the first time, the encoder registers a new entry in the Huffman tree and sends an ESCAPE code followed by the bit-string (of that character). On receipt of an ESCAPE code, a new entry relating to the character following the ESCAPE code is created by the decoder. The second option of implementation of the initialization routine is that each possible character of the input set is initialized with a constant value of 1. When scanning of the input string takes place, the recurring character will require fewer characters and vice-versa. But what makes this scheme rather inefficient is the presence of unused code.
The update_model performs two functions. Firstly, it increments the count relating to each character in the Huffman tree. It should be remembered that the initial count of a character is either a 0 or 1 based upon the initialization scheme. Secondly, it updates the Huffman tree. What necessitates updating of the tree is a change in the frequency count of a character. Change also takes place in the position of that character in the Huffman tree.

The Huffman tree keeps intact its sibling property during the updates. A binary code tree has the sibling property if each non-root node has a sibling and if the nodes can be listed in the order of non-increasing weight with each node next to its sibling. When swapping becomes essential, the farthest node with weight W is swapped with the node whose weight has just been increased to W+1.

**Example of Adaptive Huffman Coding: Algorithm FGK**

Faller [26] and Gallager [33] independently introduced essentially the same algorithm, a few years apart. It was later improved upon by Cormack and Horspool [20]. The same improvements appeared again one year later, this time proposed by Knuth [50]. The dynamically changing Huffman codes are maintained by both sender (encoder) and receiver (decoder) in algorithm FGK. The characters seen thus far and whose weights are the current frequency counts for the characters are represented by the leaves of the Huffman code trees. To begin with, the tree contains only the 0-node, a special node which represents messages that are yet to be seen. At any given stage in the execution of the algorithm, ‘k’ out of the possible ‘n’ different characters have been seen, so there are ‘k+1’ leaves in the tree, one for each of the ‘k’ seen characters plus one more for the 0-node.
The encoding and decoding part of FGK algorithm is represented in Figure 2.10 and 2.11 respectively.

```
Initialize tree with just 0-node
for each character ‘ch’
  if ‘ch’ has been seen then
    transmit ‘ch’ code
    increment ‘ch’’s frequency
    rearrange the tree if necessary
  else
    transmit the code for the 0-node
    transmit ‘ch’
    split the 0-node using one sibling
    for ‘ch’ and the other for the new 0-node
    frequency for ‘ch’ node is set to 1
    rearrange the tree if necessary
  endif
end (for)
```

**Figure 2.10: FGK Encoding Algorithm**

```
Initialize tree with just 0-node
for each codeword w
  if w is not the code for the 0-node then
    transmit w's character
    increment w's frequency
    rearrange the tree if necessary
  else
    transmit the character ‘ch’ appearing
    next in the encoding
    split the 0-node using one sibling
    for ‘ch’ and the other for the new 0-node
    frequency for ‘ch’ node is set to 1
    rearrange the tree if necessary
  endif
end (for)
```

**Figure 2.11: FGK Decoding Algorithm**
2.1.5 ARITHMETIC CODING

Huffman and Shannon-Fano coding techniques suffer from the fact that an integral value of bits is needed to code a character. Arithmetic coding completely bypasses the idea of replacing every input symbol with a codeword. Instead, it replaces a stream of input symbols with a single floating point number as output. The basic concept of arithmetic coding was first proposed by Peter Elias in the early 1960’s and was first described in [2]. It was further developed largely by Pasco[59], Rissanen [62,63] and Langdon[53]. The idea of the arithmetic coding that is generally known nowadays was independently presented in 1979 and 1980 by Rissanen and Langdon [63], Rubin [67], and Guazzo [36]. A summary of Rissanen and Langdon’s work on the field of the arithmetic coding can be found in the work of Langdon [53]. Some of the interesting works on this topic were also presented by Witten et al. [97] and Howard and Vitter [41,42,43]. A modern approach to the arithmetic coding is described in the work of Moffat et al. [56].

The primary objective of Arithmetic coding is to assign an interval to each potential symbol. Later this interval is assigned a decimal number. The algorithm commences with an interval of 0.0 and 1.0. The interval is subdivided into a smaller interval, based on the proportion to the input symbol’s probability, after each input symbol from the alphabet is read. This subinterval then becomes the new interval and is divided into parts according to probability of symbols from the input alphabet. This is repeated for each and every input symbol. And, at the end, any floating point number from the final interval uniquely determines the input data.
Figure 2.12 illustrates the Arithmetic Encoding process.

AriEnc (message)

CurInt = [0,1)

While the end of the message is not reached
   Read the letter \( x_i \) from the message;
   Divide CurInt into subintervals \( IR_{\text{CurInt}} \);
   CurInt = subinterval \( i \) in \( IR_{\text{CurInt}} \);
End While
Output bits uniquely identifying CurInt;

Figure 2.12: Arithmetic Coding – Encoding Process

Example 2.4

Let us consider an alphabet consists of symbols \( S = \{X, Y, Z, \#\} \) and \( P = \{.4, .3, .1, .2\} \). We are encoding the string XYYYY#. Here \( X \) is the first letter of the message and it is also the first letter in the set \( S \), from the CurInt (Current Interval) [0,1), the first subinterval (0,.4) is selected. Next the next letter \( Y \) is read and it is the second letter in the set \( S \). Thus the second sub interval becomes

\[
IR_{[L,R]} = \{ [L, L+(R-L) * P_1), [L+(R-L) * P_1, L+(R-L) * P_2), [L+(R-L) * P_2, L+(R-L) * P_3)],......., [L+(R-L) * P_{n-1}, L+(R-L))] \}
\]

\[
= [0+.4-.0]*.4, 0 + (.4 - 0) * (0.4 + 0.3)) = [.16, .28) \text{ of the CurInt [0,.4) is selected.}
\]

The third letter of the message is \( Y \), which also causes the second sub interval of the CurInt to be chosen as
\[.16 + (.28 - .16) \times .4, .16 + (.28 - .16) \times (.4 + .3)) = [.208, .244]\]

After reading fourth letter of the message Z, CurInt is equal to
\[.208 + (.244 - .208) \times (.4 + .3), .208 + (.244 - .208) \times (.4 + .3 + .1)) = [.2332, .2368]\]

and after reading the fifth letter # the subinterval becomes,
\[.2332 + (.2368 - .2332) \times (.4 + .3 + .1), .2332 + (.2368 - .2332) \times (.4 + .3 + .1 + .2)) = [.23608, .2368)\]

This is the process of subdividing CurInt into subintervals. Now a codeword is output that uniquely identifies CurInt. It can be any number from this interval. Normally, it is the left boundary (.23608 in our example) or the arithmetic mean:
\[(.23608 + .2368) / 2 = .23644.\]

The steps are summarized in the table in the Table 2.1.

<table>
<thead>
<tr>
<th>Current Interval</th>
<th>Length</th>
<th>Input letter</th>
<th>Sub-intervals</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,1)</td>
<td>1</td>
<td>X</td>
<td>[0, .4)</td>
</tr>
<tr>
<td>[0,4)</td>
<td>P₁ = .4</td>
<td>Y</td>
<td>[0, 16)</td>
</tr>
<tr>
<td>[.16,28)</td>
<td>P₁P₂ =.12</td>
<td>Z</td>
<td>[.16,208)</td>
</tr>
<tr>
<td>[.208,.244)</td>
<td>P₁P₂P₂ =.036</td>
<td>#</td>
<td>[.208,.2224)</td>
</tr>
<tr>
<td>[.2332,.2368)</td>
<td>P₁P₂P₂P₃ =.0036</td>
<td>#</td>
<td>[.2332,.23464)</td>
</tr>
<tr>
<td>[.23608,.2368)</td>
<td>P₁P₂P₂P₃P₄ =.00072</td>
<td></td>
<td>[.23608,.2368)</td>
</tr>
</tbody>
</table>

**Table 2.1: Divisions of the Current Interval during Encoding of the Message XYYZ#**

Here the decoding process is the same as the encoding process, but instead of using the symbols to narrow the interval, we use given interval to select a symbol, and then narrow it. The main problem with decoding is that it is not clear what stream of letters the codeword represents. This can be solved by encoding a special symbol called End of Message Marker (# in our example).
The decoding algorithm is given in Figure 2.13.

```
AriDec(codeword)
    CurInt = [0,1);
    While(1)
        Divide CurInt into subintervals IR_{CurInt};
        Determine the subinterval, of CurInt to which
        codeword belongs;
        Output letter x_i corresponding to this subinterval;
        If x_i = ‘#’
            return;
        CurInt = subinterval, in IR_{CurInt};
    End while
```

*Figure 2.13: Arithmetic Coding – Decoding Process*

**Example 2.5**

Using the source S and the probabilities P, the codeword .23608 can be decoded as follows:

First the number .23608 causes the first subinterval [0,.4), of the initial interval [0,1) to be chosen. Since it is the first subinterval, letter X is the output. Now CurInt is set to [0,.4), and in the second iteration, it is identified that the codeword .23608 belongs to the second interval of the CurInt [0,.4), which corresponds to the second letter of the source S, the letter Y. The steps are summarized and presented in Table 2.2.
<table>
<thead>
<tr>
<th>Current Interval</th>
<th>Output letter</th>
<th>Sub-intervals</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,1)</td>
<td>X</td>
<td>[.4,.7) [7,.8) [.8,1)</td>
</tr>
<tr>
<td>[0,.4)</td>
<td>Y</td>
<td>[.16,.28) [.28,.32) [.32,.4)</td>
</tr>
<tr>
<td>[.16,.28)</td>
<td>Y</td>
<td>[.28,.208) [.208,.244) [.244,.256) [.256,.28)</td>
</tr>
<tr>
<td>[.208,.244)</td>
<td>Z</td>
<td>[.2224,.2332) [.2332,.2368) [.2368,.242)</td>
</tr>
<tr>
<td>[.2332,.2368)</td>
<td>#</td>
<td>[.23464,.23572) [.23572,.23608) [.23608,.2368)</td>
</tr>
</tbody>
</table>

Table 2.2: Decoding the Codeword .23608

The arithmetic encoder output is a very long floating-point number which involves a floating point multiplication which is time consuming. Luckily, a technique is available for progressive transmission of an interval that corresponds to input data which used fixed precision arithmetic. If allowances are made for loss of some effectiveness, fixed precision integers for the arithmetic coding can also be used effectively. Round off errors in division eventually results in the loss in the integer implementation of the arithmetic coding. It must be said that that integer version is faster than floating point arithmetic coding. The loss incurred in this process is acceptably small.

The compression ratio reached by the arithmetic coding and the unconditional entropy of the input data are the same theoretically. In practice, what makes the coding less efficient is the use of integer arithmetic and scaling in order to prevent overflow of the variables which store the symbol frequencies. Nonetheless, better compression effectiveness is achieved by the arithmetic coding than by the Huffman coding. However, the arithmetic coding is slower and involves more computation. The arithmetic coding is found to be most suitable for adaptive compression, in which probabilities of input symbols may vary at each step. The Huffman coding has an edge over arithmetic coding and hence it is
preferred in a static and semi-static compression in which the probabilities of the input symbols are fixed [14,55].

Like the Huffman coding, the arithmetic coding is scarcely used as an independent compression method. It is invariably used as the coding method in the last stage of lossless data compression.

The arithmetic coding is used in connection with, for example, a PPM algorithm, BWT based algorithms, JPEG-LS [95], JBIG, Run-Length Encoding, Move-To-Front coding. It is also used in the hardware-implemented fax protocols CCITT Group 3 [48] and CCITT Group 4 [49], which use a small alphabet with an unevenly distributed probability.

2.1.6 COMPARISON OF STATISTICAL COMPRESSION TECHNIQUES

In this section, attention is focused on comparing the performance of various Statistical compression techniques (Run Length Encoding, Shannon-Fano coding, Huffman coding, Adaptive Huffman coding and Arithmetic coding). Research works done to evaluate the efficiency of any compression algorithm are carried out having two important parameters. One is the amount of compression achieved and the other is the time used by the encoding and decoding algorithms. The practical performance of the above mentioned techniques on files of Calgary corpus have been tested several times and the results of various Statistical coding techniques selected for this study have been found out [71,72]. Also, the comparative functioning and the compression ratio are presented in the tables given below.

Table 2.3 shows the comparative analysis between various Statistical compression techniques discussed above.
<table>
<thead>
<tr>
<th>S.No</th>
<th>File names</th>
<th>File Size in bytes</th>
<th>RLE</th>
<th>Shannon - Fano coding</th>
<th>Huffman coding</th>
<th>Adaptive Huffman coding</th>
<th>Arithmetic coding</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>bpc</td>
<td>bpc</td>
<td>bpc</td>
<td>bpc</td>
</tr>
<tr>
<td>1.</td>
<td>bib</td>
<td>1,11,261</td>
<td>8.16</td>
<td>5.56</td>
<td>5.26</td>
<td>5.24</td>
<td>5.23</td>
</tr>
<tr>
<td>3.</td>
<td>book2</td>
<td>6,10,856</td>
<td>8.16</td>
<td>5.08</td>
<td>4.83</td>
<td>4.83</td>
<td>4.78</td>
</tr>
<tr>
<td>4.</td>
<td>news</td>
<td>3,77,109</td>
<td>7.98</td>
<td>5.41</td>
<td>5.24</td>
<td>5.23</td>
<td>5.19</td>
</tr>
<tr>
<td>5.</td>
<td>obj1</td>
<td>21,504</td>
<td>7.21</td>
<td>6.57</td>
<td>6.45</td>
<td>6.11</td>
<td>5.97</td>
</tr>
<tr>
<td>6.</td>
<td>obj2</td>
<td>2,46,814</td>
<td>8.05</td>
<td>6.50</td>
<td>6.33</td>
<td>6.31</td>
<td>6.07</td>
</tr>
<tr>
<td>7.</td>
<td>paper1</td>
<td>53,161</td>
<td>8.12</td>
<td>5.34</td>
<td>5.09</td>
<td>5.04</td>
<td>4.98</td>
</tr>
<tr>
<td>8.</td>
<td>paper2</td>
<td>82,199</td>
<td>8.14</td>
<td>4.94</td>
<td>4.68</td>
<td>4.65</td>
<td>4.63</td>
</tr>
<tr>
<td>9.</td>
<td>progc</td>
<td>39,611</td>
<td>8.10</td>
<td>5.47</td>
<td>5.33</td>
<td>5.26</td>
<td>5.23</td>
</tr>
<tr>
<td>10.</td>
<td>progl</td>
<td>71,646</td>
<td>7.73</td>
<td>5.11</td>
<td>4.85</td>
<td>4.81</td>
<td>4.76</td>
</tr>
<tr>
<td>11.</td>
<td>progp</td>
<td>49,379</td>
<td>7.47</td>
<td>5.28</td>
<td>4.97</td>
<td>4.92</td>
<td>4.89</td>
</tr>
<tr>
<td>12.</td>
<td>trans</td>
<td>93,695</td>
<td>7.90</td>
<td>5.88</td>
<td>5.61</td>
<td>5.58</td>
<td>5.49</td>
</tr>
<tr>
<td></td>
<td>Average bpc</td>
<td></td>
<td>7.93</td>
<td>5.50</td>
<td>5.27</td>
<td>5.21</td>
<td>5.15</td>
</tr>
</tbody>
</table>

Table 2.3: Comparison of bpc for Different Statistical Compression Techniques

As per the results shown in Table 2.3, for Run Length Encoding, for most of the files tested, this algorithm generates compressed files larger than the original files. This is due to the fewer amount of runs in the source file. For the other files, the compression rate is less. The average bpc obtained by this algorithm is 7.93. So, it is inferred that this algorithm can reduce on an average of about 4% of the original file. This can not be considered as a significant improvement.
Bits Per Character (bpc) and amount of compression achieved for Shannon-Fano algorithm is presented in Table 2.3. The compression ratio for Shannon-Fano algorithm is in the range of 0.60 to 0.82 and the average bpc is 5.50.

Compression ratio for Huffman coding algorithm falls in the range of 0.57 to 0.81. The compression ratio obtained by this algorithm is better compared to Shannon-Fano algorithm and the average bits per character is 5.27.

The amount of compression achieved by applying Adaptive Huffman coding is shown in Table 2.3. The adaptive version of Huffman coding builds a statistical model of the text being compressed as the file is read. From Table 2.3 it can be seen that, it differs a little from the Shannon-Fano coding algorithm and Static Huffman coding algorithm in the compression ratio achieved and the range is between 0.57 and 0.79. On an average, the number of bits needed to code a character is 5.21. Previous attempts in this line of research make it clear that compression and decompression times are relatively high for this algorithm because the dynamic tree used in this algorithm has to be modified for each and every character in the source file.

Arithmetic coding has been shown to compress files down to the theoretical limits as described by Information theory. Indeed, this algorithm proved to be one of the best performers among these methods based on compression ratio. It is clear that the amount of compression achieved by Arithmetic coding lies within the range of 0.57 to 0.76 and the average bits per character is 5.15.

The overall performance in terms of average bpc of the above referred Statistical coding methods are shown in Figure 2.14.
Figure 2.14: Chart Showing Compression Rates for Various Statistical Compression Techniques

The overall behaviour of Shannon-Fano coding, Static Huffman coding and Adaptive Huffman coding is very similar with Arithmetic coding achieving the best average compression. The reason for this is the ability of this algorithm to keep the coding and the modeler separate. Unlike Huffman coding, no code tree needs to be transmitted to the receiver. Here, encoding is done to a group of symbols, not symbol by symbol, which leads to higher compression ratios. One more reason is its use of fractional values which leads to no code waste.
2.2 DICTIONARY BASED COMPRESSION TECHNIQUES – LEMPEL ZIV ALGORITHMS

Arithmetic algorithms as well as Huffman algorithms are based on a statistical model, namely an alphabet and the probability distribution of a source. It is observed that there are correlations between parts of data (recurring patterns) and Dictionary coding techniques are dependent upon this observation. The replacement of those repetitions by (shorter) references to a “dictionary” containing the original forms the basis of Dictionary Coding Techniques. The primary function of the dictionary is to store string patterns seen before and the indexes are used to encode the repeated patterns.

In 1987, Bell [9] showed that there is a general algorithm for converting a dictionary method to a statistical one, and any practical dictionary compression scheme can be outperformed by a related statistical compression scheme. Dictionary-based compression techniques, however, are very fast at average compression ratio, what makes them very attractive from a practical point of view.

Several techniques are adopted by dictionary compression approaches and these techniques incorporate the structure in the data with a view to achieving a better compression. The main objective of these techniques is elimination of the redundancy of storage. A list of the most common words or phrases in a document called a dictionary is maintained by the encoder. The encoder makes use of their indices in the dictionary as output tokens. The distinct advantage is that the tokens are much shorter in comparison with the words or phrases themselves and frequent repetitions of the words and phrases in the document are seen.

The encoder reads the input string, identifies those words which recur quite often and outputs their indices in the dictionary. A new word is output in the uncompressed form and added into the dictionary as a new entry. The comparison of strings, maintenance of dictionary and encoding in an efficient way are the primary processes of the dictionary-based compression methods.
A dictionary is maintained separately by compressors and decompressors. The dictionary-based algorithms are generally faster than entropy-based ones.

On the basis of how the dictionary is built and maintained, dictionary methods can be divided into two categories. They are

- Static dictionary
- Adaptive dictionary

**Static Dictionary**

A static dictionary makes use of the simplest forms of dictionary coding. Such a dictionary may consist of frequently occurring phrases of arbitrary length, digrams (two-letter combinations) or n-grams. An existing coding such as ASCII lends itself to develop this kind of dictionary easily by the use of previously unused codewords or extension of the length of the codewords to accommodate the dictionary entries. In static dictionary based method a predefined dictionary called static dictionary is used for encoding the text. The dictionary consists of codewords for most frequently occurring words. Once the dictionary is created, it could be kept online and used by both the encoder and decoder. Dictionary does not get changed while the data is being compressed. The model of static dictionary-based methods is shown in Figure 2.15. A static dictionary achieves a little compression for most data sources.

![Figure 2.15: General Static Compression](image)
There are both advantages and disadvantages in a static dictionary. The greatest advantage is that it can be fine-tuned to fit the data it is compressing but on the disadvantage side this method usually requires two passes over the data, one to build the dictionary and another one to compress the data. Moreover, transmission of the mapping determined in the first pass of a static encoding scheme must be carried out by the encoder to the decoder which results in certain amount of overhead added to the compressed form. The second disadvantage of the dictionary is that it relies upon implementation.

**Adaptive Dictionary**

There is a sharp difference between the static and adaptive dictionary which is also called dynamic dictionary. In contrast to the static dictionary, adaptive dictionary accommodates itself to changes in input data characteristics over time. Unlike the static dictionary, which has completely defined dictionary when compression begins, the schemes of adaptive dictionary begin either with no dictionary or with a default baseline dictionary. Some adaptive methods adapt to changing patterns in the source while others exploit locality of reference. The basis for the methods that adapt to change in pattern is the assumption that there is a uniform distribution of words occurring in the input data while locality of reference-based adaptation takes into account the fact that a wide variety of text has a tendency that for a particular word to occur frequently for short periods of time and then to remain unused for longer periods.
The model of adaptive dictionary-based methods is shown in Figure 2.16.

Figure 2.16: General Adaptive Compression

An adaptive dictionary is built in a single pass and the encoding of the data also takes place simultaneously since the decoder can build up the dictionary in the same method as the encoder. While decompressing the data it is not required to explicitly transmit/store the dictionary. The speed improvement in the adaptive case is the implication of the fact that an initial scan is not needed. The salient advantage of adaptive model is its ability to adapt itself to local conditions; but the main drawback of the adaptive model is that it is completely ignorant of the data when it starts its process. So in the beginning of the encoding these methods do not compress very well. The Lempel Ziv algorithm belongs to this category of dictionary coders.

The methods of dictionary-based compression can be further sub-classified into two main divisions: implicit and explicit. The objective of the first division is to try to see, if the character sequence that is currently subject to compression has already figured earlier in the input data and then output only a pointer to the earlier occurrence instead of repeating it. The dictionary created thus can be called implicit because it is represented by the previously processed data. A noteworthy point here is that the basis of all the methods of this group is founded on the algorithm developed and published in 1977 by Abraham Lempel and Jacob Ziv.
Another dictionary of the phrases that appears in the input data is created by the algorithms of the second group. When they come across a phrase which figures in the dictionary, they merely output the index number of the phrase in the dictionary. The bases of these methods are founded on the algorithm developed and published by Abraham Lempel and Jacob Ziv in 1978.

The Lempel Ziv Algorithm is an algorithm for lossless data compression. This algorithm is an offshoot of the two algorithms proposed by Jacob Ziv and Abraham Lempel in their landmark papers in 1977 and 1978. Figure 2.17 represents diagrammatically the family of Lempel Ziv algorithms.

![Figure 2.17: The Family of Lempel Ziv Algorithms](image)

2.2.1 LZ77

Jacob Ziv and Abraham Lempel have presented their dictionary-based scheme in 1977 for lossless data compression [99]. Today this technique is much remembered by the name of the authors and the year of implementation of the same.

LZ77 exploits the fact that words and phrases within a text file are likely to be repeated. When there is repetition, they can be encoded as a pointer to an
earlier occurrence, with the pointer accompanied by the number of characters to be matched. It is a very simple adaptive scheme that requires no prior knowledge of the source and seems to require no assumptions about the characteristics of the source.

In the LZ77 approach, the dictionary functions merely as a portion of the previously encoded sequence. The examination of the input sequence is carried out by the encoder, pressing into service a sliding window which consists of two parts: a search buffer that contains a portion of the recently encoded sequence and a look-ahead buffer that contains the next portion of the sequence to be encoded. An example of the window is represented in Figure 2.18.

![Sliding Window Diagram](image)

**Figure 2.18: Sliding Window**

The algorithm searches the sliding window for the longest match with the beginning of the look-ahead buffer and outputs a reference (a pointer) to that match. It is possible that there is no match at all, so the output cannot contain just pointers. In LZ77 the representation of the reference is always in the form of a triple \(<o,l,c>\), where ‘o’ stands for an offset to the match, ‘l’ represents length of the match, and ‘c’ denotes the next symbol after the match. A null pointer is generated as the reference in case of absence of the match (both the offset and the match length equal to 0) and the first symbol in the look-ahead buffer.

The values of an offset to a match and length must be limited to some maximum constants. Moreover, the compression performance of LZ77 mainly depends on these values. Usually the offset is encoded on 12–16 bits, so it is
limited from 0 to 65535 symbols. So, there is no need to remember more than 65535 last seen symbols in the sliding window. The match length is usually encoded on 8 bits, which gives maximum match length equal to 255.

The LZ77 encoding algorithm is depicted in Figure 2.19.

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

**Figure 2.19: LZ77 Encoding Algorithm**

In LZ77 encoding process one reference (a triple) is transmitted for several input symbols and hence it can be quite time-consuming, since there are a lot of comparisons to be performed between the lookahead buffer and the window. The decoding is much faster than the encoding in this process and it is one of the important features of this process. In LZ77, most of the LZ77 compression time is, however, used in searching for the longest match, whereas the LZ77 algorithm decompression is quick as each reference is simply replaced with the string, which it points to. Here, memory requirements are low both for the encoding and the decoding process.
LZ77 scheme can be made to function more efficiently through several ways. Efficient encoding with the triples forms the basis for many of the improvements.

**Example 2.6**

Let us consider the sequence to be encoded is …dbcsdbbebcbsbsbsbe…

Let us assume that the length of the window is 13 and the size of the look-ahead buffer is 6. The current content in the window is as follows:

\[
\begin{array}{c|c}
\text{dbcsdb} & \text{ebcsbs} \\
\end{array}
\]

The contents in the search buffer are ‘dbcsdb’ and in look-ahead buffer are ‘ebcsbs’. To find a match for ‘e’ it is looked back in the already encoded portion of the window. Since there is no match the triple \(<0,0,\text{c(e)}>\) is transmitted. The first two values in the triple shows that there is no match to ‘e’ in the search buffer and \(\text{c(e)}\) denotes the codeword for the symbol ‘e’. To encode a single character this method seems to a wasteful way. A single character has been encoded in order to move the window by one character. The contents of the window are:

\[
\begin{array}{c|c}
\text{bcbsdb} & \text{bsbss} \\
\end{array}
\]

A match for ‘b’ is to be found now. Starting from the current location, a match to ‘b’ is found at an offset of two with a length of match as one. Looking further back, another match for ‘b’ is found at an offset of four with a length of match again as one. Looking back even further, yet another match for ‘b’ is also found at an offset of seven with a length of match as four. So the string ‘bcbsb’ is encoded with the triple \(<7,4,\text{c(s)}>\), and move the window by five characters. The window now contains the following characters:
The contents in the look-ahead buffer are ‘sbssbe’. A match for ‘s’ has to be found now. Looking back in the window, a match for ‘s’ is found at an offset of one with a match length of one, and a second match at an offset of three with a match length of three. But a match length of five instead of three can be used. So, it can be encoded with the triple <3,5,c(e)>.

The LZ77 decoding algorithm can be described in Figure 2.20.

```
bebcbs sbssbe

The sliding window is maintained in the same way as encoding is.
In each step a triplet (Position, length, symbol) from the input is read.
if position = 0
  print symbol
else
  go reverse in previous output by Position characters and copy character
  wise for length symbols
  print symbol
endif
```

Figure 2.20: LZ77 Decoding Algorithm

The decoding process of LZ77 is explained here.

**Example 2.7**

It is assumed that the sequence ‘dbcsbdb’ is already decoded. The first triple to be decoded is <0,0,c(e)>. Here there is no match with the previously decoded string, so the next symbol is ‘e’. The decoded string now becomes ‘dbcsbdbe’.

```
d b c s b d b e
```
The next triple to be decoded is \(<7,4,c(s)>\). The first element of the triple tells the decoder to move the copy pointer back seven characters and copy four characters from that point. This is explained in Figure 2.21.

![Diagram](image)

Figure 2.21: Decoding of the Triple \(<7,4,c(s)>\)

The final triple to be decoded is \(<3,5,c(e)>\). The first element in the triple says move back three characters and start copying. The first three characters copied are ‘sbs’. Again the copy pointer moves once again to copy the recently copied ‘s’. Like this the next two characters ‘b’ and ‘s’ are copied. It is noted that
the match started in the search buffer but it is extended into the look-ahead buffer. This is illustrated in Figure 2.22.

![Diagram of decoding process](image)

**Figure 2.22: Decoding of the Triple <3,5,c(e)>**

There are several variations on LZ77 scheme, the best known are LZSS[86], LZH and LZB[57]. Further improvements were introduced also by Bell [8], Bell and Kulp [11], Bell and Witten [12], Gutmann and Bell [37], and Horspool [40].
2.2.2 LZSS

LZSS, which was published by Storer and Szymanski [86], removes the requirement of mandatory inclusion of the next non-matching symbol into each codeword. Their algorithm uses fixed length codewords consisting of offset and length to denote references. They propose to include an extra bit (a bit flag) at each coding step to indicate whether the output code represents a pair (a pointer and a match length) or a single symbol. The LZSS encoding algorithm is shown in Figure 2.23.


code

while( lookAheadBuffer not empty )
  get a pointer (position, match) to the longest match;
  if (length > MINIMUM_MATCH_LENGTH)
    output (POINTER_FLAG, position, length);
    shift the window length characters along;
  else
    output (SYMBOL_FLAG, first symbol of lookahead buffer);
    shift the window 1 character along;
  endif
end while

code

Figure 2.23: LZSS Encoding Algorithm

This algorithm generally yields a better compression ratio than LZ77 with practically the same processor and memory requirements. The decoding is still extremely simple and quick. That's why it has become the basis for practically all the later algorithms of this type.
2.2.3 LZH

LZH is the scheme proposed by Bernd Herd that combines the Lempel–Ziv and Huffman techniques. Here coding is performed in two passes. The first is essentially same as LZSS, while the second uses statistics measured in the first to code pointers and explicit characters using Huffman coding.

2.2.4 LZB

LZB published by Mohammad Banikazemi[57] uses an elaborate scheme for encoding the references and lengths with varying sizes. The size of every LZSS pointer remains the same despite the length of the phrase it represents. Different-sized pointers prove to be efficacious in practice as they help achieve a better compression since some phrase lengths are prone to occur more frequently than others. LZB is a technique that uses a different coding for both components of the pointer. LZB achieves a better compression than LZSS and has the added virtue of being less sensitive to the choice of parameters.

2.2.5 LZR

LZR, developed by Michael Rodeh et al. [66] in the year 1991, is a modification of LZ77. It is projected to be linear time alternative to LZ77. It is markedly different from the already existing algorithm in its capacity to allow pointers to denote any position in the encoded part of the text. However, it should be mentioned that LZR consumes considerably larger amount of memory than the others do. Here, the dictionary grows without any limit.
The two major drawbacks of this algorithm are:

- More and more memory is required as encoding proceeds; no more of the input is remembered if the memory is full or the memory should be cleared for resumption of the coding process.
- It also suffers from a drawback of the increase in the size of the text in which the matches are sought. As it is an unfeasible variant, its performance is found to be not satisfactory.

2.2.6 LZ78

In 1978 Jacob Ziv and Abraham Lempel presented their dictionary-based scheme, which is known as LZ78 [100]. It is a dictionary-based compression algorithm that maintains an explicit dictionary. This dictionary has to be built both at the encoding and decoding sides and they must follow the same rules to ensure that they use an identical dictionary. The codewords output by the algorithm consists of two elements <i,c> where ‘i’ is an index referring to the longest matching dictionary entry and the ‘c’ is the first non-matching symbol. In addition to output the codeword for storage / transmission the algorithm also adds the index and symbol pair to the dictionary. When a symbol is not yet found in the dictionary, the codeword has the index value 0 and it is added to the dictionary as well. The algorithm gradually builds up a dictionary with this method. The algorithm for LZ78 encoding is depicted in Figure 2.24.
w := NIL;
while ( there is input )
    K := next symbol from input;
    if (wK exists in the dictionary)
        w := wK;
    else
        output (index(w), K);
        add wK to the dictionary;
        w := NIL;
    endif
end while

Figure 2.24: LZ78 Encoding Algorithm

Example 2.8

The following sequence xbcebxbcbxbcbxbcbx is encoded using the LZ78 approach. Here ‘b’ represents space. Here, initially the dictionary is empty, so the first few symbols scanned are encoded with the index value of 0. The first three outputs of encoder are <0,c(x)>, <0,c(b)> and <0,c(c)>. Initially, the dictionary looks as shown in Table 2.4.

<table>
<thead>
<tr>
<th>Index</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
</tr>
</tbody>
</table>

Table 2.4: Initial LZ78 Dictionary
If it is continued like this, the encoder and the dictionary are shown in the Table 2.5.

<table>
<thead>
<tr>
<th>Encoder output</th>
<th>Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Index</td>
</tr>
<tr>
<td>&lt;0,c(x)&gt;</td>
<td>1</td>
</tr>
<tr>
<td>&lt;0,c(b)&gt;</td>
<td>2</td>
</tr>
<tr>
<td>&lt;0,c(c)&gt;</td>
<td>3</td>
</tr>
<tr>
<td>&lt;3,c(b)&gt;</td>
<td>4</td>
</tr>
<tr>
<td>&lt;0,c(\text{e})&gt;</td>
<td>5</td>
</tr>
<tr>
<td>&lt;1,c(b)&gt;</td>
<td>6</td>
</tr>
<tr>
<td>&lt;3,c(c)&gt;</td>
<td>7</td>
</tr>
<tr>
<td>&lt;2,c(\text{e})&gt;</td>
<td>8</td>
</tr>
<tr>
<td>&lt;6,c(c)&gt;</td>
<td>9</td>
</tr>
<tr>
<td>&lt;4,c(\text{e})&gt;</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 2.5: Development of LZ78 Dictionary

LZ78 Decoding

The decoding algorithm reads an element of the tokens at a time from the compressed file and maintains the dictionary in a similar way as the encoder.

Let \(<x, c>\) be a compressed token pair, where ‘x’ is the next codeword and ‘c’ the character after it.

The algorithm for LZ78 decoding is given in Figure 2.25.

```
while not EOF do
    x = next_codeword ( )
    c = next_char ( )
    output dict_word(x) + c
    add dict_word(x) + c into dictionary at the next available location
end while
```

Figure 2.25: LZ78 Decoding Algorithm
The decoding operation of LZ78 on the input tokens 0x 0b 0c 3b 0x 1b 3c 2x 6c 4x is shown step by step in the following Table 2.6.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>x</th>
<th>c</th>
<th>W(x) + c</th>
<th>Output</th>
<th>Dictionary D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td>0 1 2 3 4 5 6 7 8 9 10</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>b</td>
<td>b</td>
<td>x b</td>
<td>x</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>c</td>
<td>c</td>
<td>x b c</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>cb</td>
<td>cb</td>
<td>x b c cb</td>
<td>x</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>b</td>
<td>b</td>
<td>x b c</td>
<td>x</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>xb</td>
<td>xb</td>
<td>x b c</td>
<td>x</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>cc</td>
<td>cc</td>
<td>x b c</td>
<td>cc</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>b</td>
<td>b</td>
<td>x b c</td>
<td>x</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>c</td>
<td>c</td>
<td>x b c</td>
<td>x</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>cb</td>
<td>cb</td>
<td>x b c cb</td>
<td>x</td>
</tr>
</tbody>
</table>

**Table 2.6: LZ78 Decoding Process**

The original string xbcxbxbccxbccxbcc is obtained back.

LZ78 algorithm has the ability to capture patterns and hold them indefinitely but it also has a serious drawback. The dictionary keeps growing forever without bound. There are various methods to limit dictionary size, the easiest being to stop adding entries and continue like a static dictionary coder or to throw the dictionary away and start from scratch after a certain number of entries has been reached. The biggest advantage over the LZ77 algorithm is the reduced number of string comparisons in each encoding step. The encoding done by LZ78 is fast, compared to LZ77, and that is the main advantage of dictionary-based compression. The important property of LZ77, that the LZ78 algorithm preserves, is the decoding which is faster than the encoding.
There are several variations on LZ78 scheme, the best known are LZW[96], LZFG[29], LZC[90] and LZR[91].

2.2.7 LZW

Terry Welch presented LZW (Lempel–Ziv–Welch) algorithm in 1984[96], which is based on LZ78. It basically applies the LZSS principle of not explicitly transmitting the next non-matching symbol to LZ78 algorithm. The dictionary has to be initialized with all possible symbols from the input alphabet. It guarantees that a match will always be found. LZW would only send the index to the dictionary. The input to the encoder is accumulated in a pattern ‘w’ as long as ‘w’ is contained in the dictionary. If the addition of another letter ‘K’ results in a pattern ‘w*K’ that is not in the dictionary, then the index of ‘w’ is transmitted to the receiver, the pattern ‘w*K’ is added to the dictionary and another pattern is started with the letter ‘K’. The LZW encoding algorithm is presented in Figure 2.26.

```
w := NIL;
while ( there is input )
   K := next symbol from input;
   if (wK exists in the dictionary)
      w := wK;
   else
      output (index(w));
      add wK to the dictionary;
      w := K;
   endif
end while
```

Figure 2.26: LZW Encoding Algorithm

71
Example 2.9

Consider the previously used sequence to demonstrate LZ78 algorithm as the input sequence for LZW.

Input Sequence : xbcxbxbccbxbc

Alphabet for the source is [x, b, c, x]

Initially LZW dictionary is shown in Table 2.7.

<table>
<thead>
<tr>
<th>Index</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 2.7: Initial LZW Dictionary

Here, the encoder first scans the letter ‘x’. This pattern is available in the dictionary so the next letter to it is combined, thus forming the pattern ‘xb’. This pattern is not in the dictionary and so the letter ‘x’ is encoded with the dictionary index 4, and the pattern ‘xb’ is added into the dictionary as 5th element and a new pattern is going to begin with the letter ‘b’. As ‘b’ is found in the dictionary, the next element ‘c’ is concatenated to form the pattern ‘bc’. This is not present in the dictionary, so ‘b’ is encoded with the dictionary index 2, the pattern ‘bc’ is added to the dictionary as 6th element and start identifying a new pattern with the letter ‘c’. If it is continued like this, two letter patterns are constructed until the letter ‘x’ is reached in the second ‘xbccbx’. The next symbol read is ‘b’. Joining this with ‘x’, the pattern ‘xb’ is obtained, which is available in the dictionary as 5th entry. So, the next symbol is read which is ‘c’. Concatenating this with ‘xb’ we get ‘xbc’. This is not available in the dictionary, so it is included as the 11th entry in the dictionary and a new pattern is started with the letter ‘c’. The length of the entries continues
to increase in correspondence with the further progress of the encoding. The longer entries in the dictionary are indicative of the fact that the dictionary captures more of the structure in the sequence. At the end of the encoding process, the dictionary looks like as shown in Table 2.8.

<table>
<thead>
<tr>
<th>Index</th>
<th>Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
</tr>
<tr>
<td>4</td>
<td>x</td>
</tr>
<tr>
<td>5</td>
<td>xb</td>
</tr>
<tr>
<td>6</td>
<td>bc</td>
</tr>
<tr>
<td>7</td>
<td>cc</td>
</tr>
<tr>
<td>8</td>
<td>cb</td>
</tr>
<tr>
<td>9</td>
<td>b=b</td>
</tr>
<tr>
<td>10</td>
<td>b=x</td>
</tr>
<tr>
<td>11</td>
<td>xbc</td>
</tr>
<tr>
<td>12</td>
<td>ccb</td>
</tr>
<tr>
<td>13</td>
<td>b=b=x</td>
</tr>
<tr>
<td>14</td>
<td>xbcc</td>
</tr>
<tr>
<td>15</td>
<td>cb=b</td>
</tr>
</tbody>
</table>

Table 2.8: LZW Dictionary for Encoding

The output sequence of the encoder is as follows:

4 2 3 3 2 1 5 7 9 11 8 1

12 bits are set as the size of the pointer, making provision for up to 4096 dictionary entries. The dictionary becomes static as soon as the optimum limit of 4096 is reached.
LZW Algorithm – Decoding

LZW decoding algorithm is as shown in Figure 2.27.

```
read a token ‘x’ (code or char) from the compressed file
look up dictionary for the element ‘k’ at ‘x’
output ‘k’
w := k
while not EOF do
  read ‘x’
  look up dictionary for the element ‘k’ at ‘x’
  if there is no entry for index ‘x’ then
    k := w * firstcharof(w)
  output ‘k’
  add w * firstcharof (k) to the dictionary
  w := k
end while
```

**Figure 2.27: LZW Decoding Algorithm**

Here, the encoder output is considered from the above example and decode it using the LZW algorithm. The encoder output sequence was 4 2 3 3 2 1 5 7 9 11 8 1 which becomes the input sequence for the decoder. The decoder starts with the same initial dictionary as the encoder. (Table 2.8). The index value 4 corresponds to the letter ‘x’, so ‘x’ is decoded as the first element of the sequence. In order to mimic the dictionary construction procedure of the encoder, the construction of the next element of the dictionary is done as follows. It starts with the letter ‘x’, which exists in the dictionary. So it is not added into the dictionary and the decoding process is continued. The next decoder input is 2, which is the index corresponding to ‘b’. ‘b’ is decoded and added it to our current pattern ‘x’ to form the pattern ‘xb’. It is not in the dictionary, so it is added as the fifth element
of the dictionary and a new pattern is started with the letter ‘b’. The next input of
the decoder is 3 which is the index corresponding to the letter ‘c’. ‘c’ is decoded
and added it to our current pattern ‘b’ to form the pattern ‘bc’. Since it is not in the
dictionary, it is added as 6th entry in the table. The entire sequence can be decoded
continuing in this fashion. It is important to note that the dictionary being
constructed by the decoder is identical to that constructed by the encoder.

2.2.8 LZFG

What distinguishes LZFG which was developed by Fiala and Greene [29],
is the fact that encoding and decoding are fast and good compression is achieved
without undue storage requirements. This algorithm uses the original dictionary
building technique as LZ78 does but the only difference is that it stores the
elements in a trie data structure. Here, the encoded characters are placed in a
window (as in LZ77) to remove the oldest phrases from the dictionary.

2.2.9 LZC

Lempel Ziv Compress (LZC) developed by Thomas et al.[90] in 1985,
which finds its application in UNIX Compress utility, is a slight modification of
LZW. It has as its origin the implementation of LZW and subsequently stands
modified as LZC with the specific objective of achieving faster and better
compression. It has earned the distinction of being a high performance scheme as
it is found to be one of the most practically and readily available schemes. A
striking difference between LZW and LZC is that the latter, LZC, monitors the
compression ratio of the output whereas the former, LZW, does not. Its value lies
in its utility to rebuild the dictionary from the scratch, clearing it completely if it
crosses a threshold value.
2.2.10 LZW

Lempel Ziv Tischer (LZW) developed by Tischer [91] in 1987, is a modification of LZC. The main difference between LZW and LZC is that it creates space for new entries by discarding least recently used phrases (LRU replacement) if the dictionary is full.

Of all the LZ78 variants, LZW algorithm is easily the most popular algorithm. The best known applications of LZW are Unix Compress, GIF and V.42 bis. The deflate algorithm originally designed by Phil Katz is the most popular implementation of LZ77 algorithm. It is part of the popular zlib library developed by Jean-loup Gailly and Mark Adler. Jean-loup Gailly also used deflate in the gzip algorithm which is used widely. The deflate algorithm is also used in PNG. The UNIX compress command is one of the earlier applications of LZW. The Graphics Interchange Format (GIF), another implementation of the LZW algorithm, was developed by CompuServe Information Service to encode graphical images. It is very similar to the compress command.

2.2.11 COMPARISON OF LEMPEL ZIV ALGORITHMS

This section deals with comparing the performance of Lempel Ziv algorithms. LZ algorithms considered here are divided into two categories: those derived from LZ77 and those derived from LZ78 [71,72]. Table 2.9 shows the comparison of various algorithms that are derived from LZ77 (LZ77, LZSS, LZH, LZB and LZR). Table 2.10 shows the comparative analysis of algorithms that are derived from LZ78 (LZ78, LZW, LZFG, LZC and LZT). The bpc values that are referred from [10] are based on the following parameters. The main parameter for LZ77 family is the size of the window on the text. Compression is best if the window is as big as possible, but not bigger than the text, in general. Nevertheless,
larger windows yield diminishing returns. A window as small as 8000 characters will perform much better, and give a result nearly as good as the ones derived from the larger windows. Another parameter which limits the number of characters is needed for some algorithms belonging to LZ family. Generally, a limit of around 16 may work well. For LZ77, LZSS and LZB the storage (characters in window) were assumed to be of 8 KB, for LZH it was assumed as 16 KB and for LZR it was unbounded.

Regarding LZ78 family, most of the algorithm requires one parameter to denote the maximum number of phrases stored. For the above mentioned LZ78 schemes, except LZ78 a limit of 4096 phrases was used.

<table>
<thead>
<tr>
<th>S.No</th>
<th>File Names</th>
<th>File Size in bytes</th>
<th>LZR bpc</th>
<th>LZ77 bpc</th>
<th>LZSS bpc</th>
<th>LZH bpc</th>
<th>LZB bpc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>bib</td>
<td>1,11,261</td>
<td>3.59</td>
<td>3.75</td>
<td>3.35</td>
<td>3.24</td>
<td>3.17</td>
</tr>
<tr>
<td>2.</td>
<td>book1</td>
<td>7,68,771</td>
<td>4.61</td>
<td>4.57</td>
<td>4.08</td>
<td>3.73</td>
<td>3.86</td>
</tr>
<tr>
<td>3.</td>
<td>book2</td>
<td>6,10,856</td>
<td>3.97</td>
<td>3.93</td>
<td>3.41</td>
<td>3.34</td>
<td>3.28</td>
</tr>
<tr>
<td>4.</td>
<td>news</td>
<td>3,77,109</td>
<td>4.26</td>
<td>4.37</td>
<td>3.79</td>
<td>3.84</td>
<td>3.55</td>
</tr>
<tr>
<td>5.</td>
<td>obj1</td>
<td>21,504</td>
<td>6.37</td>
<td>5.41</td>
<td>4.57</td>
<td>4.58</td>
<td>4.26</td>
</tr>
<tr>
<td>6.</td>
<td>obj2</td>
<td>2,46,814</td>
<td>4.21</td>
<td>3.81</td>
<td>3.30</td>
<td>3.19</td>
<td>3.14</td>
</tr>
<tr>
<td>7.</td>
<td>paper1</td>
<td>53,161</td>
<td>4.47</td>
<td>3.94</td>
<td>3.38</td>
<td>3.38</td>
<td>3.22</td>
</tr>
<tr>
<td>8.</td>
<td>paper2</td>
<td>82,199</td>
<td>4.56</td>
<td>4.10</td>
<td>3.58</td>
<td>3.57</td>
<td>3.43</td>
</tr>
<tr>
<td>9.</td>
<td>progc</td>
<td>39,611</td>
<td>4.39</td>
<td>3.84</td>
<td>3.24</td>
<td>3.25</td>
<td>3.08</td>
</tr>
<tr>
<td>10.</td>
<td>progl</td>
<td>71,646</td>
<td>3.05</td>
<td>2.90</td>
<td>2.37</td>
<td>2.20</td>
<td>2.11</td>
</tr>
<tr>
<td>11.</td>
<td>progp</td>
<td>49,379</td>
<td>2.97</td>
<td>2.93</td>
<td>2.36</td>
<td>2.17</td>
<td>2.08</td>
</tr>
<tr>
<td>12.</td>
<td>trans</td>
<td>93,695</td>
<td>2.50</td>
<td>2.98</td>
<td>2.44</td>
<td>2.12</td>
<td>2.12</td>
</tr>
<tr>
<td><strong>Average bpc</strong></td>
<td></td>
<td></td>
<td><strong>4.08</strong></td>
<td><strong>3.88</strong></td>
<td><strong>3.32</strong></td>
<td><strong>3.22</strong></td>
<td><strong>3.11</strong></td>
</tr>
</tbody>
</table>

Table 2.9: Comparison of bpc for the Different LZ77 Variants
The output of Table 2.9 reveals that the Bits per Character is significant and most of the files have been compressed to a little less than half of the original size. Of LZ77 family, the performance of LZW is significant compared to LZ77, LZSS, LZH and LZR. The average bpc, which is significant as shown in Table 2.9, is 3.11.

Amongst the performance of the LZ77 family, LZW outperforms LZH. This is because, LZH generates an optimal Huffman code for pointers whereas LZW uses a fixed code.

Figure 2.28 shows a comparison of the compression rates for the different LZ77 variants.

![Image of compression rates](image.png)

**Fig 2.28: Compression Rates for the LZ77 Family**
<table>
<thead>
<tr>
<th>S.No</th>
<th>File Names</th>
<th>File Size in bytes</th>
<th>LZ78 bpc</th>
<th>LZW bpc</th>
<th>LZFG bpc</th>
<th>LZC bpc</th>
<th>LZW bpc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>bib</td>
<td>1,11,261</td>
<td>3.95</td>
<td>3.84</td>
<td>2.90</td>
<td>3.89</td>
<td>3.76</td>
</tr>
<tr>
<td>2.</td>
<td>book1</td>
<td>7,68,771</td>
<td>3.92</td>
<td>4.03</td>
<td>3.62</td>
<td>4.06</td>
<td>3.90</td>
</tr>
<tr>
<td>3.</td>
<td>book2</td>
<td>6,10,856</td>
<td>3.81</td>
<td>4.52</td>
<td>3.05</td>
<td>4.25</td>
<td>3.77</td>
</tr>
<tr>
<td>5.</td>
<td>obj1</td>
<td>21,504</td>
<td>5.58</td>
<td>6.30</td>
<td>4.03</td>
<td>6.15</td>
<td>4.93</td>
</tr>
<tr>
<td>6.</td>
<td>obj2</td>
<td>2,46,814</td>
<td>4.68</td>
<td>9.81</td>
<td>2.96</td>
<td>5.19</td>
<td>4.08</td>
</tr>
<tr>
<td>7.</td>
<td>paper1</td>
<td>53,161</td>
<td>4.50</td>
<td>4.58</td>
<td>3.03</td>
<td>4.43</td>
<td>3.85</td>
</tr>
<tr>
<td>8.</td>
<td>paper2</td>
<td>82,199</td>
<td>4.24</td>
<td>4.02</td>
<td>3.16</td>
<td>3.98</td>
<td>3.69</td>
</tr>
<tr>
<td>9.</td>
<td>progc</td>
<td>39,611</td>
<td>4.60</td>
<td>4.88</td>
<td>2.89</td>
<td>4.41</td>
<td>3.82</td>
</tr>
<tr>
<td>10.</td>
<td>Progl</td>
<td>71,646</td>
<td>3.77</td>
<td>3.89</td>
<td>1.97</td>
<td>3.57</td>
<td>3.03</td>
</tr>
<tr>
<td>11.</td>
<td>Progp</td>
<td>49,379</td>
<td>3.84</td>
<td>3.73</td>
<td>1.90</td>
<td>3.72</td>
<td>3.09</td>
</tr>
<tr>
<td>12.</td>
<td>trans</td>
<td>93,695</td>
<td>3.92</td>
<td>4.24</td>
<td>1.76</td>
<td>3.94</td>
<td>3.46</td>
</tr>
<tr>
<td></td>
<td>Average bpc</td>
<td>4.26</td>
<td>4.90</td>
<td>2.89</td>
<td>4.37</td>
<td>3.81</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.10: Comparison of bpc for the Different LZ78 Variants

It is inferred from Table 2.10 the compression performance of LZ78 family. Most of the ASCII files are compressed to just less than half of the original size and within each file the amount of compression is consistent. The LZW method, having no boundary, accepts phrases and so the compression expands the file ‘obj2’ by 25%, which is considered as a weakness of this approach. Also from Table 2.10 it is obvious that the performance of LZFG is the best amongst these methods, giving an average bpc of 2.89 which is really significant. Amongst LZ78 family, LZFG’s performance is the best because the scheme that it uses is carefully selected codes to represent pointers which are like the best scheme in the LZ77 family. Figure 2.29 represents a comparison of the compression rates for the LZ78 family.
2.3 PREDICTIVE COMPRESSION TECHNIQUES

It is acknowledged that an enhanced compression is realized when the data to be coded possesses a more skewed set of probabilities i.e., some of the symbols happen to occur with much higher probability when compared with others. This situation can be arrived by transforming the chain of symbols into another form which possesses the preferred property. In such cases, it is necessary to intimate the information regarding the transformation function to the decoder. Instead, if the transformation is attained based on the history of the sequence of symbols, then the necessity to pass on the supplementary information can be eliminated.

Following that path is the arrival of Predictive techniques. Since the history of the sequence is exploited in a predictive manner in order to establish its encoding, these schemes are recognized as Predictive schemes. The core concept behind Predictive compression techniques is to make use of the benefit of the
previous set of characters in order to generate a conditional probability of the present symbol. The application of conditional probabilities for the encountered symbols facilitates attaining higher efficiency. The usage of arithmetic encoding as one of the stages in its two-stage compression, makes the predictive coding techniques to be considered as a subset of statistical coding.

2.3.1 PPM

Cleary and Witten [18] formerly developed the “Prediction by Partial Match” method. Later with some augmentations, it was implemented by A. Moffat[54]. This method is competent enough to attain excellent compression on a wider class of data. It was stated by Cleary and Witten that PPM was capable of achieving compression in such a way that, the English text can be represented by as low as 2.2 bits / character.

PPM is an adaptive statistical data compression technique in which the statistical model of the text is rooted on an encoder. Based on the number of times every symbol has been encountered in the past, it is assigned a probability and based on this probability, the source data is encoded by the arithmetic encoder. Together with the frequency, the past context is also taken into consideration in case of context-based statistical model. In the context model, a fixed progression of symbols foregoing the current symbol is called the context and the corresponding length of the progression of the symbols is called the order of the context. Relevant update on the count of the symbols’ emergence is maintained in the context model. In the same manner, PPM order is the maximum allowable order of the context. The context in which the algorithm exists, while it encodes a new input symbol is called ‘active’. Being ‘r’ as the PPM model’s order, the order of the active context lies between zero and ‘r’. PPM model is updated after the new symbol is encoded. The contexts, if they do not exist, are created then.
PPM order is inspired by quite a few factors. Larger memory requirements but improved compression efficiency are the associated consequences of higher order. To switch the PPM model to a shorter context, a new ‘escape’ symbol is added to each context. PPMA [18], PPMB [18], PPMC [54], PPMD [45], PPMZ [13], and PPMII [80] are some of the vital methods for selecting the probability of the escape symbol.

The major demerits of PPM algorithm such as high memory requirements, comparatively low compression speed are diminished by raising the maximum context order. But this, in turn, amplifies the computational necessity.

2.3.2 PPM*

Cleary et al. [19] formulated some enhancements in PPM to end up with PPM*. In the primary PPM method, one of the vital features is the utilization of fixed-length, bounded initial context i.e., it starts up with a value N as the context length and the next symbol is always expected beginning with an order N context C. If the symbol has not been found hitherto, then the algorithm moves to some shorter context. In the case of PPM*, the value of N is extended indefinitely. This results in a new-fangled data structure and further more computational resources, when compared with that of PPM. This additional pay results in 6% more upgrading of compression than PPMC, the method used for assigning escape probabilities.

The algorithm for PPM* is based on the concept of deterministic context, which gives only one prediction. A pointer to the previous context’s appearance for each deterministic context is preserved by PPM* algorithm. The probability of a new symbol is much lesser than anticipated in such contexts. During the process of encoding, shortest deterministic context is captured from the list of already
existing contexts. A pointer to a reference context is contained in this context. A predicted symbol is the initial symbol following reference context, in the earlier stage. If accurate prediction is attained, then the predicted symbol is encoded by PPM* followed by encoding a new predicted symbol. The escape symbol is used in case of failure of prediction when PPM* acts like a normal PPM coder.

When applied to the compression of Calgary corpus, PPM* comes up with an outcome of an average of 2.34 bpc which is 2.48 bpc for PPMC.

2.3.3 PPMZ

Charles Bloom [13] in 1998 implemented PPMZ as an effort to enrich the original PPM algorithm. PPMZ tries to cope with the traits such as deterministic contexts, unbounded-length contexts and local order estimation which result in less favorable performance of PPM and succeed in attaining a cut above execution. Like PPM*, this method also starts up with the inspection of present symbol’s maximum deterministic context. If any deterministic context is not found, then a Local-Order-Estimation (LOE) procedure is carried out as the next step.

The secondary escape estimation is another crucial attribute of PPMZ for the estimation of escape probabilities. In this mechanism, the statistics of occurrences of escape symbols for similar PPM contexts are joined together. This model consists of order 2 escape contexts, which is independent from PPM model. The entities such as PPM order, number of escapes and finally the number of correct predictions are combined together to generate the escape context.

PPMZ has its foundation on PPM* for the unbounded length deterministic contexts. But it is present with some variations. Very lengthy unbounded contexts (order 12 or higher) and minimum match length are employed by PPMZ in order
to eradicate errors. In addition to that the unbounded contexts are deterministic constantly at all the time.

Also, supplementary to the standard PPM model are the unbounded contexts in PPMZ which are in a way independent of it. A pointer to the last context’s appearance, the minimum match length and the match frequency are sustained by each unbounded context. In PPMZ, the unbounded context models are composed adaptively. Following the encoding of each symbol, the unbounded context models are constructed which are then indexed by a hashed value of the last 12 occurred symbols.

During the process of encoding, a proper unbounded deterministic context for an active context is located by the algorithm and a pointer to a reference context is held by the unbounded context. An order of common context for the active context and the reference context is then computed by PPMZ. PPMZ checks if the match length is equal to or greater than the minimum match length (which initially equals 12). PPMZ acts in a similar way as ordinary PPM, in case either the unbounded deterministic context is not found or the match length is too small.

PPMZ encoding is carried out otherwise i.e., when the length is either equal to or greater than the minimum match length for this context. In case the correct symbol is predicted, it is encoded and the match frequency in the unbounded deterministic context is updated by PPMZ algorithm. The process is repeated as a search for the next unbounded deterministic context which matches with a new active context. The escape symbol is released if the prediction is not successful and ordinary PPM encoding is applied. Thus, PPMZ forces all unbounded contexts to be deterministic.

PPMZ, when put into operation for Calgary corpus, attains an average of 2.119 bpc which is 10% more than PPM* and 17% more than PPMC.
2.3.4 Fast PPM

Howard and Vitter [44] developed and implemented fast PPM in 1994 as an alternative to PPM. The out of favor state of PPM is that it is slow. To speed up its progression, those factors that only marginally facilitate to realize performance of compression are substituted by approximations in Fast PPM.

Modeling and Encoding are the two major features of PPM. To figure out the probability of recent symbol, the modeling part investigates the contexts of the symbol. This process, which is complex, is reduced to bare bones by eradicating the plain usage of escape symbols, approximate probabilities are figured out and the exclusion procedure is made simpler in fast PPM. Quasi Arithmetic coding is employed for encoding against adaptive arithmetic coding of PPM which, in turn, enhances the speed. In the process of encoding, Fast PPM encodes each character by encoding a series of NFs (not founds) which is then succeeded by one F (found). For still more intensifying the speed of the process, for symbols with maximum probability, the quasi-arithmetic coder is brought into play to encode the NFs only. To encode the symbol’s location in the rest of the list, Rice code is aptly utilized.

On observing the compression ratio of Fast PPM on Calgary corpus, it is apparent that this method achieves 2.341 bpc which is lesser than the 2.074 bpc attained by PPMC. But when we focus our attention on the speed, the speed of Fast PPM is about 25,000 – 30,000 characters/second which is 16,000 characters/second for PPMC.