Lempel-Ziv (LZ77) Algorithm

1. Running time of the encoder (compression)

To avoid any confusions, I will be referring to different parts of the of encoding parameters as in the figure 1.1. The parameters which the Lempel-Ziv (LZ77) encoding depends on are the Search Buffer (SB) size and the Lookahead Buffer (LA) size. As data is read as bytes, the program that I implemented allows to set the parameters for SB size and LA size an integer between 0 and 2^32 which can be represented in (1-3 Bytes respectively). Setting a size of 255 for example would allow to loop through 255 bytes in the selected window and it would be stored in the encoded sequence as 1 byte.

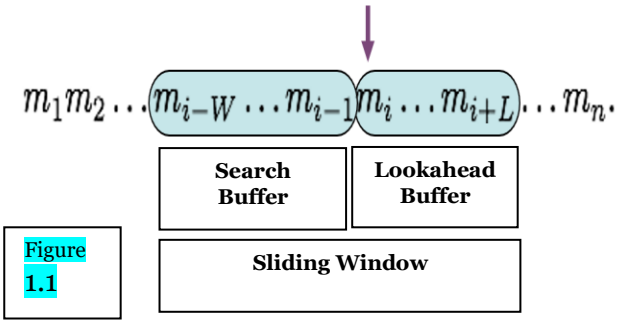


Figure 1.1

In the following example I will be analysing the running time of the encoder with the 2 parameters ***Search Buffer Size*** and ***Lookahead Buffer size*** on a sample bitmap picture *'logo.bmp'* (included in the submission folder) of original size 10.4 Kbytes.

The 2 experiments consist of setting the size of one of the windows static and dynamically change the size of the other window.

As we can see from *Figure 1.2* and *Figure 1.3*, increasing the either of the Search buffer of lookahead buffer with increase the running time, but as we will later discuss (it also increases the compression ratio). In the example of the dynamic LA size we can see that increasing the LA size from 10 to 1000 has caused an increase in time of **~43.7%**, whilst for the dynamic SB size increasing the window size caused and in increase in time of **~76%.**

An optimal encoding is obtained when a SB size and LA size of 255 (stored as 1 Byte) is used for relatively small files (up to 10kBytes) and as seen from future experimentation a SB size of **65535** (stored as 2 Bytes) and LA size of **255** (stored as 1 Byte) is optimal for large file (>10kBytes).

Search Buffer Size	***Lookahead Buffer Size***	Running Time of encoder (in seconds)
100	10	0.00798
100	100	0.00998
100	1000	0.00948
100	10000	0.01147

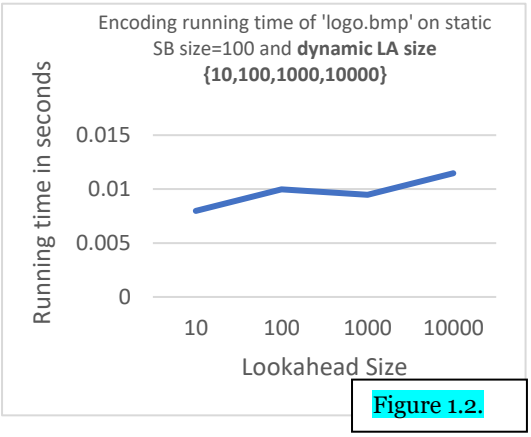


Figure 1.2.

Search Buffer Size	***Lookahead Buffer Size***	Running time of encoder (in seconds)
10	100	0.00848
100	100	0.01098
1000	100	0.01198
10000	100	0.01497

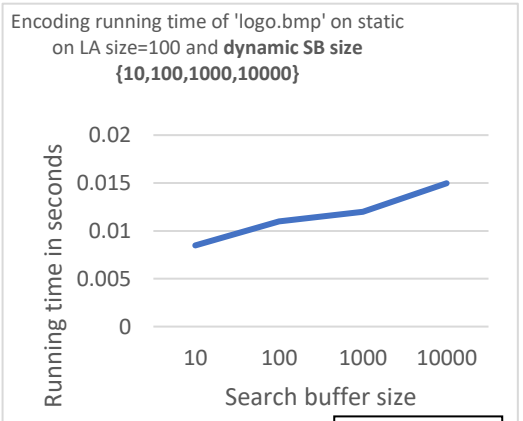


Figure 1.3.

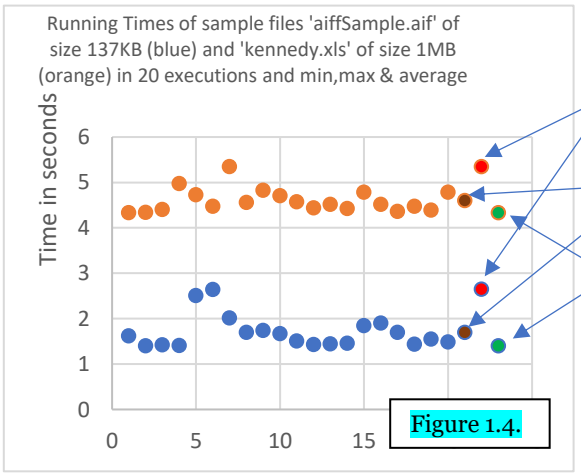


Figure 1.4.

The following experiment involves analysing the average/minimum/maximum running time for two files (.aif file of size 137 Kbytes and .xls file of size 1MB). Analysing the smaller file, when running the program 20 times we can observe that running times for this particular file vary at most 1.2s from each other. The maximum running time recorded was 2.642s, minimum 1.4s and average of 1.694s as seen from *Figure 1.4*. Comparing this to a larger file (kennedy.xls of size 1MByte), the running times similarly vary at most 1.014s, but the average running time increases by 2.7 times. This is ok as the files have different compression ratios and the second file (kennedy.xls) is about 7.3 times bigger than the first one (aiffSample.aif).

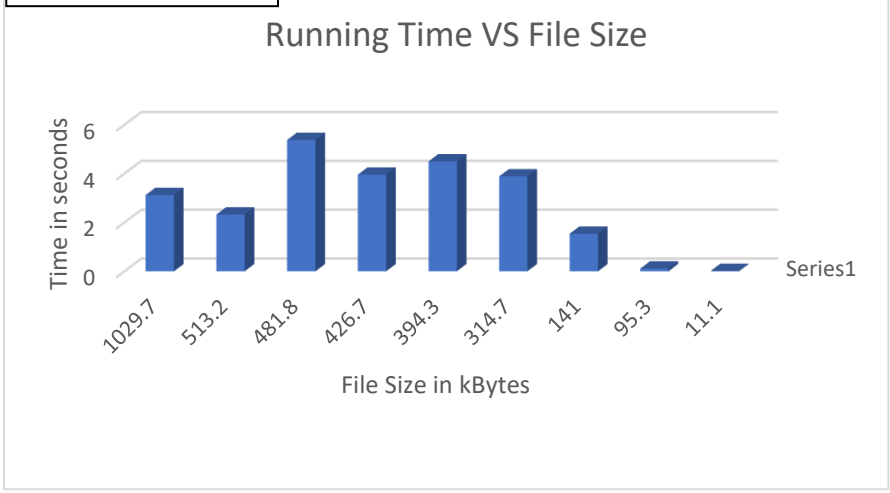
Lempel-Ziv (LZ77) Algorithm

Experimentation with different file types and sizes. Some of test files below are taken from The Canterbury Corpus. All the test files below have been tested using SB 65535 and LA 255 (2 Bytes and 1 Byte).

From *Table 1.5.* we can observe that smaller files generally execute faster than larger files, but this is not the only determinant which affects the running time. Files where data varies more and is less repetitive take longer to execute. As we can see from the table although the excel spreadsheet *'kennedy.xls'* is considerably bigger than the poem *'plrabn12.txt'*, the poem takes longer to execute, because it's less redundant in terms of its content (this can be justified by looking at the compression ratio 1.7 for poetry VS 3.12 the excel spreadsheet). In *Figure 1.6* shows that running the encoding algorithm on two files of similar size ~370Kb and getting 2 very different running times. This happened because files with more repetitive data compress faster (file 1 has compression ratio **0.65** and running time 7.85s and file 2 has compression ratio **1.82** and running time 4.2s)

File Type	Category	File name	File Size (in Kbytes)	Running Time (in seconds)
.xls	Excel Spreadsheet	kennedy.xls	1029.7	3.123
File (fax)	CCITT test set	ptt5	513.2	2.327
.txt	Poetry	plrabn12.txt	481.8	5.381
.txt	Technical writing	lcet10.txt	426.7	3.960
.bmp	Bitmap Image	boy.bmp	394.3	4.506
.wav	Audio file	tango.wav	314.7	3.890
.aif	AIF file	aiffSample.aif	141	1.541
.bmp	Black & white bitmap image	binary.bmp	95.3	0.113
.c	C source	fields.c	11.1	0.017

Table & Figure 1.5.



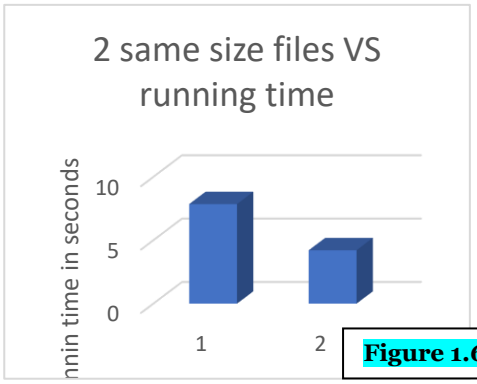


Figure 1.6

2. Running time of the decoder (decompression)

For the decompression experiments I will use the same window sizes and file that I used in part1 for encoder. The first experiment will involve setting one of the ***SB*** and ***LA size*** static and the other one dynamically allocated from the set {10,100,1000,1000}. So, the file I will be using is 'logo.bmp'. Similarly, as in the encoding part, the decoder running time increases as the LA size increases. From the *Figure 2.1* we can observe that the decoding time sees an increase of about **22.9%** when increasing the LA buffer from 10 to 10000**.** In comparison with the encoding part, it increases about half as slow.

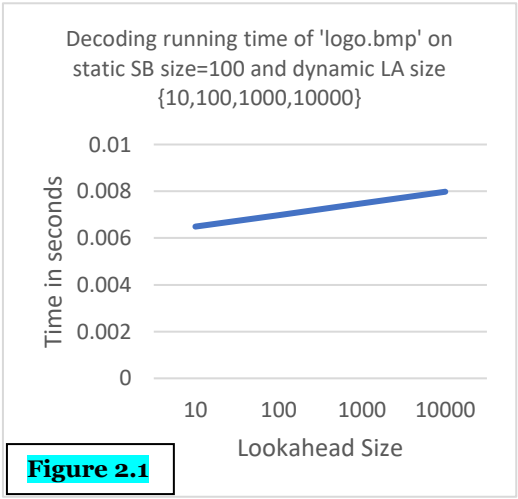


Figure 2.1

Search Buffer size	***Look-ahead buffer size***	Running time of decoder (in seconds)
100	10	0.00649
100	100	0.00698
100	1000	0.00748
100	10000	0.00798

Table 2.1

Lempel-Ziv (LZ77) Algorithm

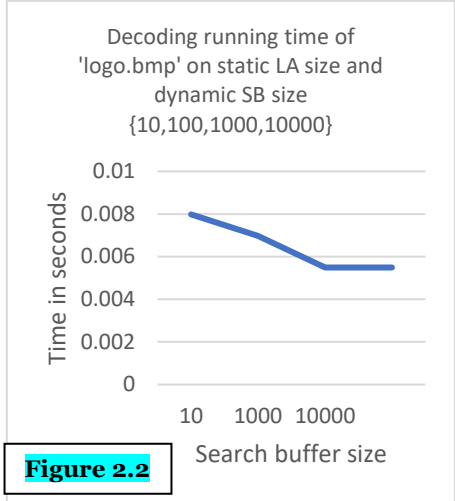


Figure 2.2

Search Buffer size	***Look-ahead buffer size***	Running time of decoder (in seconds)
10	100	0.00798
100	100	0.00898
1000	100	0.00549
10000	100	0.00549

Table 2.2

When increasing the search buffer size from 10 to 10000, the running time of the decoder decreases by about **31.2%**. Increasing the search buffer size allows a wider window for parsing and hence the possibility of finding longer matches and more characters to be compressed into fewer bytes, hence the decreased running time.

Average, Min and Max running times *(Figure 2.3)*

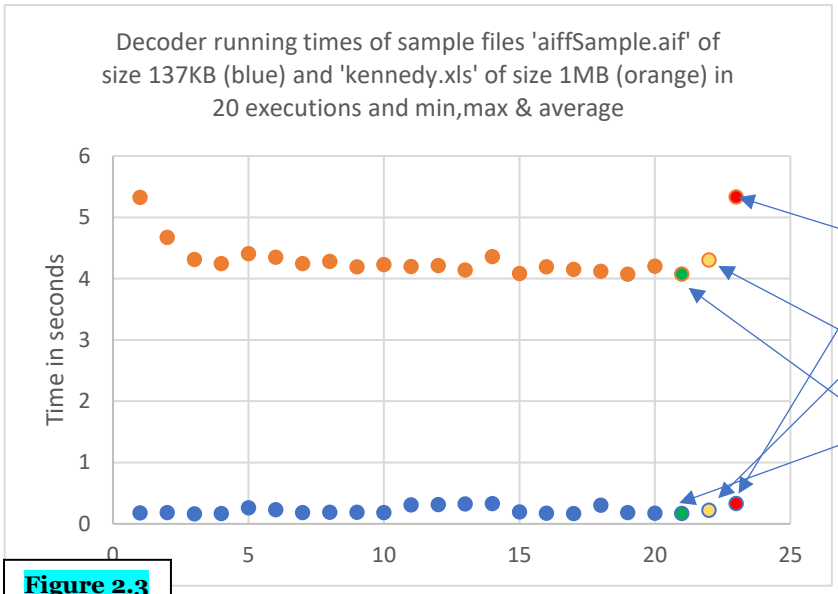


Figure 2.3

For comparison reasons with the encoder, I will use the same files as in encoding part for decompressing now. The following experiment involves analysing the average/minimum/maximum decoding running time for two files (.aif file of size 137 Kbytes and .xls file of size 1MB). We can observe that the running times on the bigger file differ by at most of 1.2s and only 0.17s for the smaller file. Also, we can notice that the decoder took much shorter to decompress the smaller file than to compress it (1.69s VS 0.22s). With the bigger file it took about the same amount of time to encode and decode, that's because comparing to the smaller file, the bigger file has a smaller compression ratio, hence, longer to decode it.

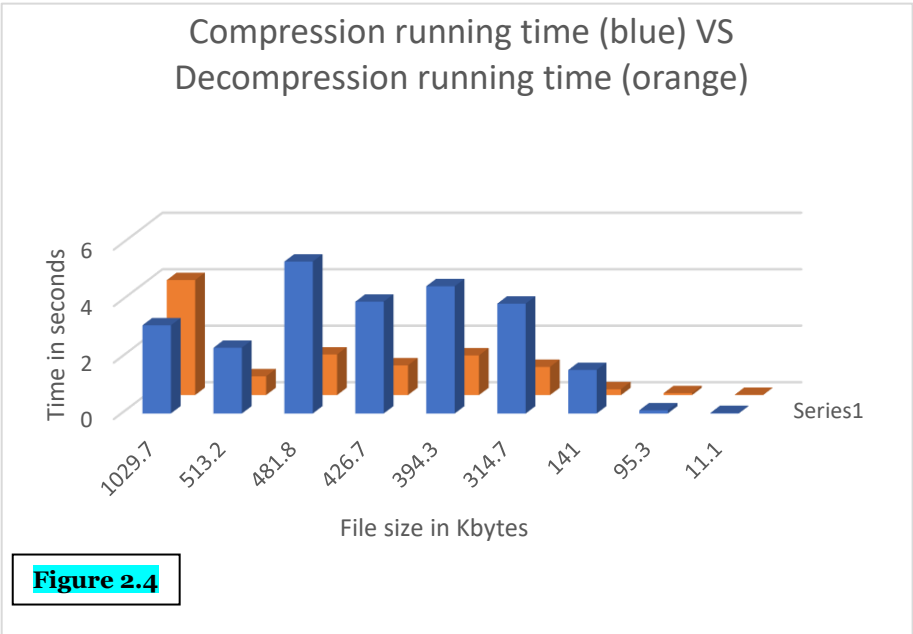


Figure 2.4

Compression running time VS Decompression running time

Generally, the encoder takes longer to execute than the decoder as we can see from *Figure 2.4*. It sometimes might be the case that the decoder takes longer to execute in the larger files with smaller compression ratios. The files analysed in this example are the same as in *Table 1.5*.

Lempel-Ziv (LZ77) Algorithm

3. Compression ratio (size of original file divided by size of compressed file)

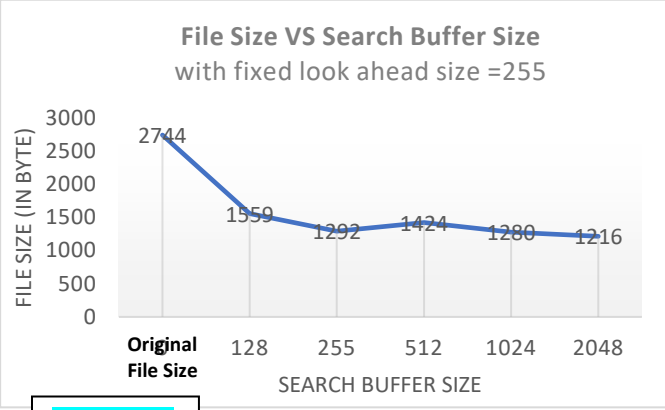


Figure 3.1

The file analysed in this example is russian.txt which contains a tongue-twister in Russian. At the cost of storing the search buffer size value in 2 bytes (when greater than 255) we can see that when the search buffer reaches 512 it slightly goes up compared to the previous value (stored in 1 byte) but with higher look ahead size it does eventually go further down than the best result obtained with storing the search buffer size in 1 byte. So, having a higher search buffer size does help with compressing the file better.

Analysing the dynamic lookahead buffer size, we can notice that having bigger values can actually increase the file size because the normally the biggest match lengths are in the range (0,255), so no need to allow another byte for this length. The conclusion we can draw from this is that increasing the search buffer size produces optimal results when combined with a (<255) lookahead size. In the next experiments I will be using the SB=65535 (2 Bytes) and LA=255 (1 Byte)

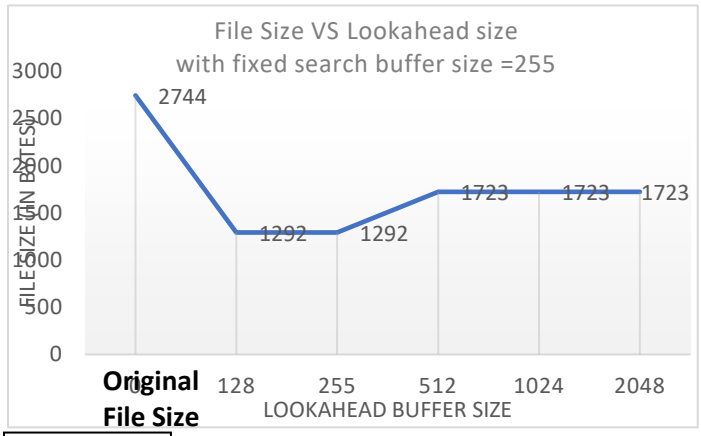


Figure 3.2

Lempel-Ziv Experiments:

File Extension	File Name	Original Size (in Bytes)	Compressed Size (in Bytes)	Compression Ratio
TXT	peterPiper.txt	193	102	1.89
TXT	bible.txt	4,047,392	421,071	9.61
TXT (in Russian)	russian.txt	2,744	1,216	2.25
BMP (picture)	logo.bmp	10,422	2,051	5.08
BMP (picture)	binary.bmp	95,310	3,779	25.2
WAV (audio)	Tango.wav	31,4784	313,703	1.0034
AIF (audio)	aiffSample.aif	141,094	104,379	1.35
XLS (spreadsheet)	kennedy.xls	1,029,744	329,659	3.123
C (Cprogram)	fields.c	11,150	5,559	2.005
LSP	grammar.lsp	3,721	2,415	1.540
JPG	flower.jpg	108,683	171,715	0.63

Table 3.1

Experimentation with different file types and sizes

From the results obtained in *Table 3.1* we can see that the Lempel Ziv algorithm achieves relatively good compression when applied on non-compressed file. For demonstration purposes I included a JPG file as well. As we can observe the JPG file didn't compress the file, in fact it made it larger. That's because JPG, PNG, MP3, MP4 etc are files that are pre-compressed and use very good compression techniques. So, we can't do better than that. Whilst if we have a look at the sample files that don't have pre-compression applied like BMP, TXT, WAV, C, XLS, AIF etc Lempel-Ziv manages to achieve compression ratios as high as **25.2** (binary.bmp in *Table 3.1*). The average compression ratio in files presented in Table 3.1 is **5.3** (not including the JPG file).

Like mentioned before the main feature that determines the efficiency of the Lempel-Ziv is how similar is the data processed from the files and how repetitive. Lempel-Ziv uses dictionary coders that's why it relies on data repetition.

Lempel-Ziv (LZ77) Algorithm

4. Comparison with other compression techniques (taken off-the-shelf or coded by yourself)

For the comparison algorithm, I chose to use the python **zlib** C library which is an abstraction of the DEFLATE compression algorithm, used for data compression.

The first experiment in *Figure 4.1.* consists of comparing the compression ratios produced by the Lempel-Ziv algorithm that I implemented versus the zlib. As we can observe in 9/10 sample files used, the zlib outperformed the Lempel-ziv algorithm by generating higher compression, however LZ77 proves to perform the best for the largest sample file used 'bible.txt' (compression ratio of 9.61 VS 3.43). Nevertheless, Zlib is a winner in this experiment.

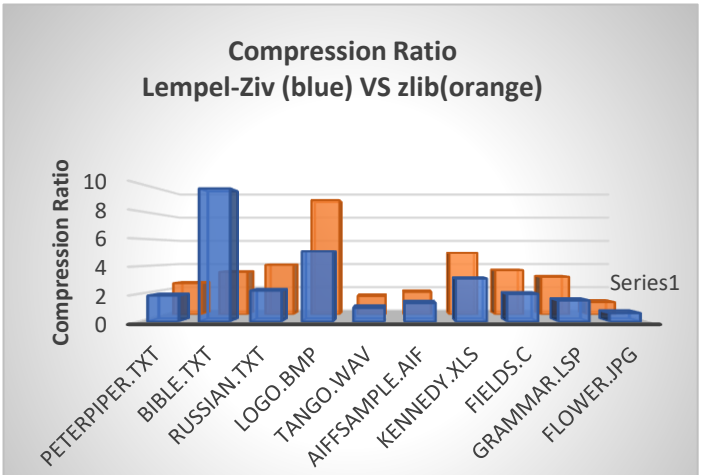


Figure 4.1

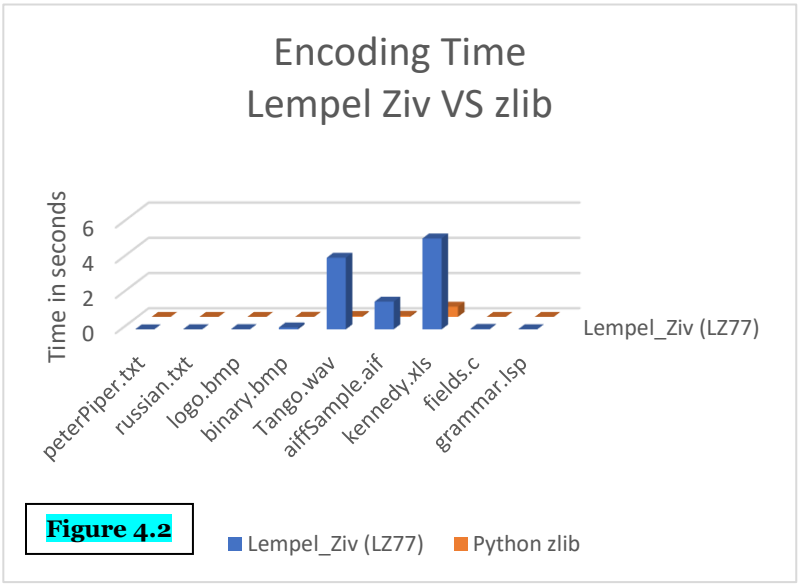


Figure 4.2

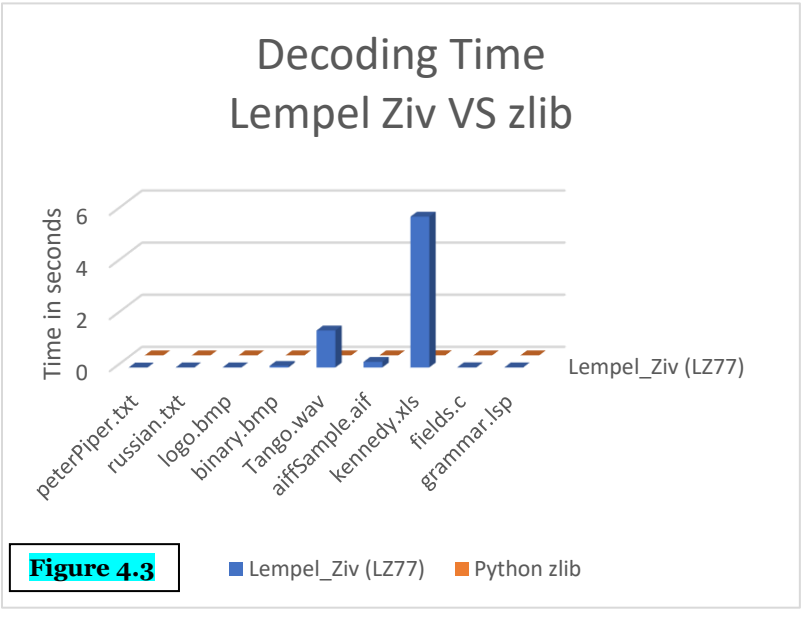


Figure 4.3

The second experiment preformed (*Figure 4.2*) demonstrates the encoding time taken by both algorithms and again zlib outperforms Lempel-Ziv. We can notice that in 6/9 sample files used, the difference in performance is not significant, and only when we get to process larger files, zlib tends to do vastly better. Similarly, when analysing the time for decompression (*Figure 4.3)* we get similar performances in 7/9 file samples. However, when testing it with larger file size Lempel-Ziv underperforms. **Conclusion:** Although LZ77 generally manages to achieve good performance with compressible files, there are other algorithms like zlib for instance that outperforms in terms of compression ratio, encoding and decoding time.

File Name	LZ77 Compression Ratio	Python zlib Compression Ratio
peterPiper.txt	1.89	2.53
bible.txt	9.61	3.43
russian.txt	2.25	4.0
logo.bmp	5.08	9.21
binary.bmp	25.2	44.5
Tango.wav	1.0034	1.49
aiffSample.aif	1.35	1.82
kennedy.xls	3.123	4.97
fields.c	2.005	3.57
grammar.lsp	1.540	3.04
flower.jpg	0.63	1.00

Table 4.1