

# **Assignment 6: “Lempel-Ziv Compression”**

## **WRITEUP**

Vincent Liu

3/9/23

Prof. Veenstra

---

### **1. Introduction**

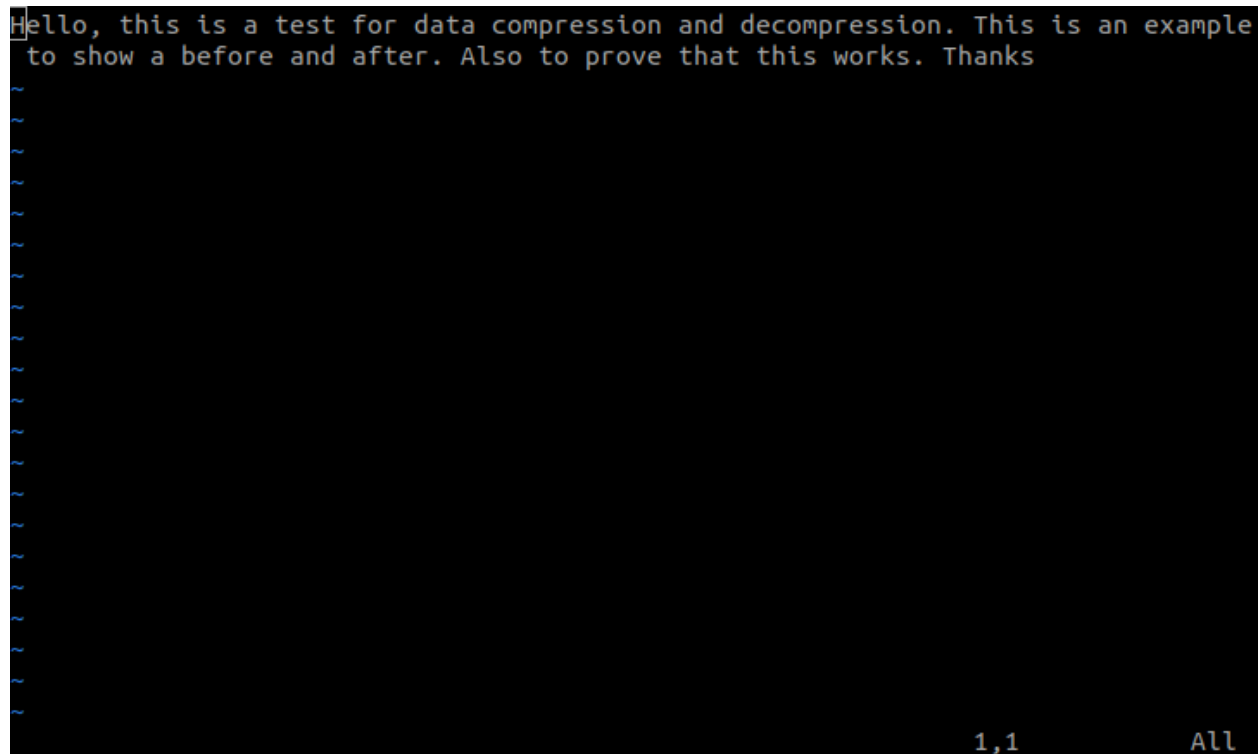
Assignment 6, titled “Lempel Ziv Compression” introduces students to the concept of data compression. The purpose of this assignment is for students to achieve a basic understanding of data compression using the Lempel-Ziv algorithm of compression (also known as LZ77 and LZ78, published in 1977 and 1978, respectively). Students will be working on the implementation of LZ78, a lossless data compression algorithm, using the C programming language and its libraries. The compression algorithm is to “...represent repeated patterns in data using pairs which are each composed of a code and a symbol. Two main files are to be created from this assignment that compress and decompress binary and/or text files. Three supporting function files are also required in order to complete the assignment. The main files are encode.c and decode.c, while the other three files are io.c, word.c and trie.c. On top of compression and decompression, interoperability between little and big endian systems is expected. Read and Write blocks are expected to be in efficient blocks of 4KB. Both encode (compress) and decode (decompress) are to use variable bit-length codes. Students will be working with a structure named TrieNodes (to implement Tries), which offers simplicity in the implementation of tries. On top of this, word tables will also be utilized, which, in essence, is a “dictionary for letters”. Each letter will be saved into a struct in conjunction with a symbol that represents said letter.

---

### **2. Example of Data Compression and Decompression (encode/decode)**

In this section I will be presenting a before and after compressing and decompressing a text file. This is simply an example of what can be seen when running the executable properly. I will be presenting the input, output, verbose output, as well as the encoded output of the encode executable.

### Input Data: Text File (input.txt)



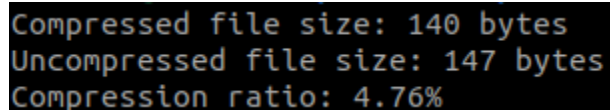
Hello, this is a test for data compression and decompression. This is an example to show a before and after. Also to prove that this works. Thanks

1,1 All

**Figure 1**

In this figure is my input text, written in a '.txt' file. Note that this is barely enough to be compressed such that the compressed file is smaller than the uncompressed version. Any text file compressed with the encode executable containing less characters than this will result in a compressed file larger than the original uncompressed version. This is because of how encode adds a header and padding.

### Verbose output of compressing 'input.txt'



```
Compressed file size: 140 bytes
Uncompressed file size: 147 bytes
Compression ratio: 4.76%
```

**Figure 2**

This figure shows the verbose output of the encode and decode executables after compression and decompression. As stated in the description above for figure 1, the text file is barely large enough to be compressed to be smaller than the uncompressed file. In this figure we are able to verify that, as the uncompressed file is 147 bytes, while the compressed file is 140 bytes, resulting in a compression ratio of 4.76%.

### Encoded Data:

```
00000000  ac ba ad ba b4 81 00 00 21 55 16 36 be 85 25 20 | .....!U.6..% |
00000010  41 17 68 91 16 73 97 b6 20 11 76 74 63 0e 81 04 | A.h..s.. .vtc...|
00000020  b3 f0 16 e4 0e 99 a3 73 20 61 6c b6 05 38 5a b6 | .....s al..8Z. |
00000030  e6 d4 5b 70 3b 61 1e 59 55 86 bd 05 6d 9a 1c 34 | ..[p;a.YU...m..4|
00000040  d7 b9 05 2e 07 55 92 d6 a4 35 61 1e c8 80 e7 b4 | .....U...5a.....|
00000050  69 6c 03 08 f2 76 cc 25 6f c1 dd 07 12 88 0d 66 | i l...v.%o.....f|
00000060  93 1c 1c 06 82 38 66 48 19 e5 72 04 11 73 13 90 | .....8fH..r..s..|
00000070  18 08 e9 2d 60 07 a3 4b 84 45 e8 52 73 87 3b dc | ...-`..K.E.Rs.;. |
00000080  da c2 05 85 e6 70 0b ac 2d 14 00 00 | .....p..-... |
0000008c
```

**Figure 3**

This figure shows the compressed output of the “encode” executable. In this figure we are able to see the magic number “0xBAADBAAC” in little endian.

### Output Data: Text file (output.txt)

```
vincent@vincent:~/Documents/cse13s/asgn6$ ./decode -i output.txt
Hello, this is a test for data compression and decompression. This is an example
to show a before and after. Also to prove that this works. Thanks
```

**Figure 4**

This figure shows the decompressed output of the “decode” executable. As expected, it is exactly the same as that of our input. We are also able to see the command used to decode the output result of the encode executable.

---

## 3. Functions used in this assignment

This assignment requires students to create three main files (written in c) in conjunction with three supporting ‘.c’ files as well as a Makefile. The functions included in these files will be elaborated on below. Note: The steps taken in each of the main files will not be elaborated on here, but within the DESIGN.pdf.

### 3.1 encode

This is the main file that is used to compress either a text or binary file into a file that is a significant amount of bits smaller than its original size.

This file uses getopt to take in arguments when running the executable. Valid arguments are as follows:

- -v : Print compression statistics to stderr.
- -i : Specify input to compress (stdin by default)

- -o: Specify output of compressed input (stdout by default)
- -h : displays program synopsis and usage.

### 3.2 decode

This is the main file that is used to decompress a file that was compressed by the encode executable.

This file uses getopt to take in arguments when running the executable. Valid arguments are as follows:

- -v : Print decompression statistics to stderr.
- -i : Specify input to decompress (stdin by default)
- -o : Specify output of decompressed input (stdout by default)
- -h: Displays program synopsis and usage.

### 3.4 trie.c

This file creates a trie. A trie is similar to a trie, however a trie contains a sequence in its structure that is used for data retrieval. In this assignment we will be using a struct for this.

- **TrieNode \*trie\_node\_create(uint16\_t code)**
  - Constructor for a TrieNode.
- **void trie\_node\_delete(TrieNode \*n)**
  - Destructor for a TrieNode.
- **TrieNode \*trie\_create(void)**
  - Initializes a trie: a root TrieNode with the code EMPTY\_CODE.
- **void trie\_reset(TrieNode \*root)**
  - Resets a trie to just the root TrieNode.
- **void trie\_delete(TrieNode \*n)**
  - Deletes a sub-trie starting from the trie rooted at node n.
- **TrieNode \*trie\_step(TrieNode \*n, uint8\_t sym)**
  - Returns a pointer to the child node representing the symbol sym.

### 3.5 word.c

Contains functions that create, edit and delete a Words and WordTables. Word holds an array of symbols, and is also referred to as WordTable

- **Word \*word\_create(uint8\_t \*syms, uint32\_t len)**
  - Constructor for a word
- **Word \*word\_append\_sym(Word \*w, uint8\_t sym)**

- Constructs a new Word from the specified Word, *w*, appended with a symbol, *sym*.
- **void word\_delete(Word \*w)**
  - Destructor for a Word, *w*.
- **WordTable \*wt\_create(void)**
  - Creates a new *WordTable*, which is an array of Words.
- **void wt\_reset(WordTable \*wt)**
  - Resets a WordTable, *wt*, to contain just the empty Word.

### 3.6 io.c

Contains functions that read and write bytes, pairs and symbols for an efficient encode and decode program.

- **int read\_bytes(int infile, uint8\_t \*buf, int to\_read)**
  - Used to read a specified amount of data from infile
- **int write\_bytes(int outfile, uint8\_t \*buf, int to\_write)**
  - Used to write a specified amount of data from infile
- **void read\_header(int infile, FileHeader \*header)**
  - This reads in sizeof(FileHeader) bytes from the input file.
- **void write\_header(int outfile, FileHeader \*header)**
  - Writes sizeof(FileHeader) bytes to the output file.
- **bool read\_sym(int infile, uint8\_t \*sym)**
  - An index keeps track of the currently read symbol in the buffer.
- **void write\_pair(int outfile, uint16\_t code, uint8\_t sym, int bitlen)**
  - “Writes” a pair to outfile.
- **void flush\_pairs(int outfile)**
  - Writes out any remaining pairs of symbols and codes to the output file.
- **bool read\_pair(int infile, uint16\_t \*code, uint8\_t \*sym, int bitlen)**
  - “Reads” a pair (code and symbol) from the input file.
- **void write\_word(int outfile, Word \*w)**
  - “Writes” a pair to the output file.
- **void flush\_words(int outfile)**
  - Writes out any remaining symbols in the buffer to the outfile.

### 3.7 Makefile

This file is used to compile the program. It also creates object files (‘.o’ files) and the necessary executable files for this assignment. The executable files created by Makefile for this assignment include: ‘keygen’, ‘encrypt’, and ‘decrypt’. The object files for this assignment are as follows:

- trie.o
- word.o
- io.o
- decode.o
- encode.o

The Makefile also properly formats all ‘.c’ files to the c17 standard. Additionally, the Makefile includes the ‘pkg-config’ required for the simplicity of library inclusion for files within this assignment.

---

## 4. Commands used in terminal

### **./encode**

This is the main file that is used to compress either a text or binary file into a file that is a significant amount of bits smaller than its original size.

This file uses getopt to take in arguments when running the executable. Valid arguments are as follows:

- -v : Print compression statistics to stderr.
- -i : Specify input to compress (stdin by default)
- -o: Specify output of compressed input (stdout by default)
- -h : displays program synopsis and usage.

### **./decode**

This is the main file that is used to decompress a file that was compressed by the encode executable.

This file uses getopt to take in arguments when running the executable. Valid arguments are as follows:

- -v : Print decompression statistics to stderr.
- -i : Specify input to decompress (stdin by default)
- -o : Specify output of decompressed input (stdout by default)
- -h: Displays program synopsis and usage.

- **make / make all** : Used to create executable files as well as compile the necessary ‘.c’ files into objects files (‘.o’ files)
- **make clean** : Used to remove the executable main files as well as remove the associated object files
- **make decode**: Used to build only the decode program
- **make encode**: Used to build only the encode program
- **make format** : Used to properly format all ‘.c’ files.

### **hexdump -C “output.txt”**

Used to get the compressed output of encode executable. Used to create figures 3, 7, and 10.

- The “-C” part of the command displays the Canonical hex+ASCII display. The man page states, “Display the input offset in hexadecimal, followed by sixteen space-separated, two-column, hexadecimal bytes, followed by the same sixteen bytes in %\_p format enclosed in | characters.”
- 

## **5. Files in the directory**

- **encode.c:**
  - This contains the main() function for the encode program.
- **decode.c:**
  - This contains the main() function for the decode program
- **trie.c:**
  - the source file for the Trie ADT
- **trie.h:**
  - the header file for the Trie ADT.
- **word.c:**
  - the source file for the Word ADT
- **word.h:**
  - the header file for the Word ADT.
- **io.c:**
  - the source file for the I/O module.
- **io.h:**
  - the header file for the I/O module
- **endian.h:**
  - the header file for the endianness module.

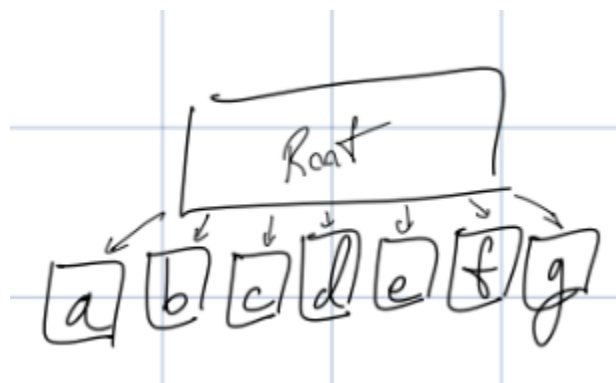
- **code.h:**
  - the header file containing macros for reserved codes.
- **Makefile**
  - Used to clean directory, generate associated executable files, and properly format .c files.
- **README.md**
  - Text file in Markdown format that describes how to build and run the program.
- **DESIGN.pdf**
  - covers the purpose of the program
  - layout/structure of the program
- **WRITEUP.pdf**
  - Describes what the program does, gives insight on the results found, explains what was learned, and clarifies steps taken to complete the assignment.

---

## 6. Example of WordTable (simplified)

For this assignment, I first had to learn how the Lempel-Ziv Compression worked. In particular, LZ78, a lossless data compression algorithm developed in 1978. How this particular data compression algorithm works is that a specific amount of data from the input file (or stdin) is read in, and its data placed into a trie. Each character read in is given a corresponding symbol before reading another segment of data from the file. If a character follows another character that already exists within the trie, it will be appended below it as a branch/leaf and given a symbol. For better understanding of this concept, (and for clarity) the following figure will show how a trie looks after a file containing “abcdefgabcdddeeeeeee” is read in.

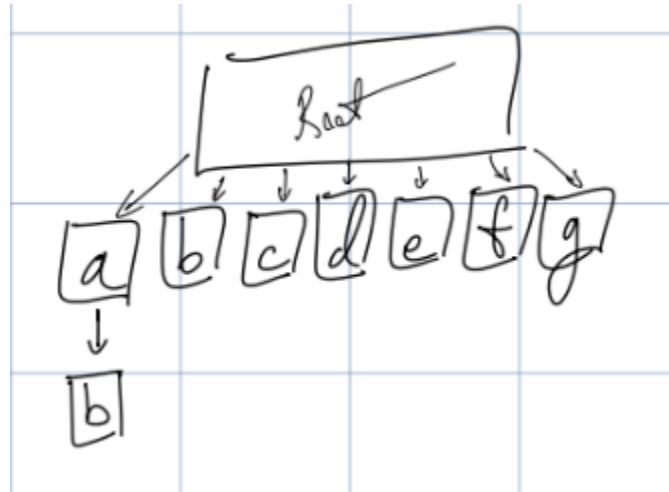
**Note: the following figures depict a highly simplified version of the WordTable used within this assignment. It is simplified to better understand what this program does.**





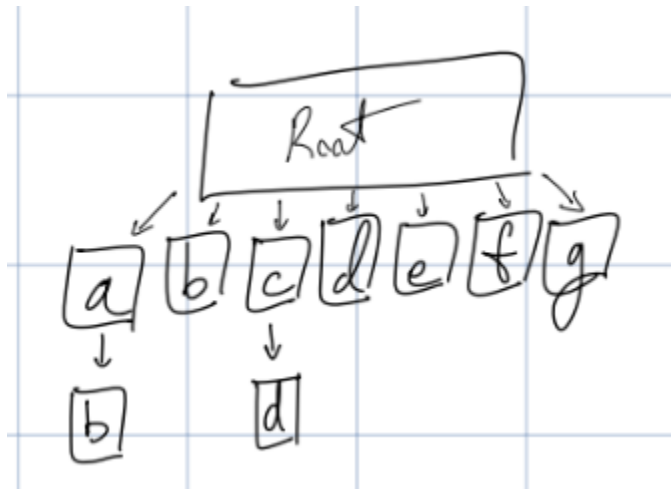
### Step 1:

In this figure, the first chunk of new characters within the text is read in and added to the struct. Since it is the first occurrence of each character in “abcdefg” to appear, each will be appended separately. We will refer to these first characters as the first level.



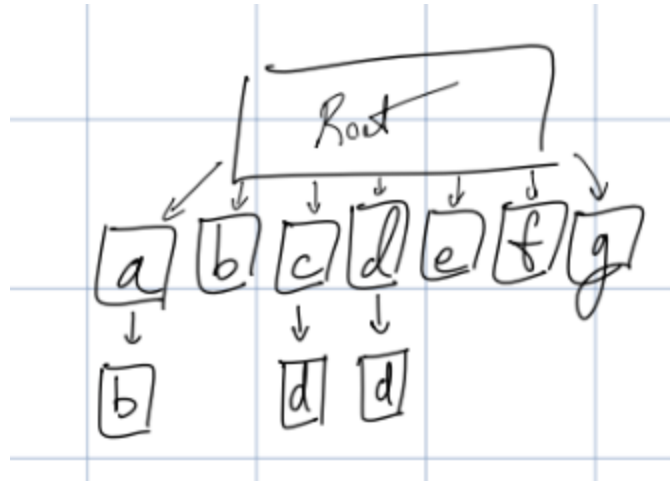
### Step 2:

In step two we append “abc” to the struct. We append these characters one at a time. Since ‘a’ already exists amongst the first level, we need not to write ‘a’ again, thus we will travel down the path of ‘a’ and append ‘b’ to it. Because ‘b’ is a new addition under first level ‘a’, we are forced to take a step back from it, back to the root.



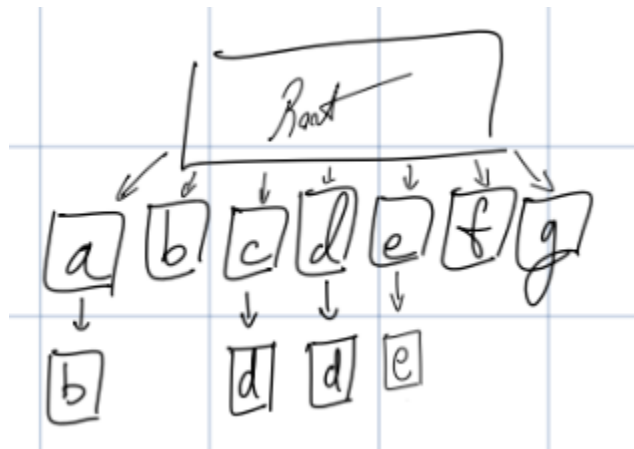
### Step 3:

In step 3 we add “cd” to the struct. We will do something similar to what was done in step 2. Since ‘c’ already exists in the first level, we will add ‘d’ under it. Because a new character was appended, we are forced to take a step back to the root.



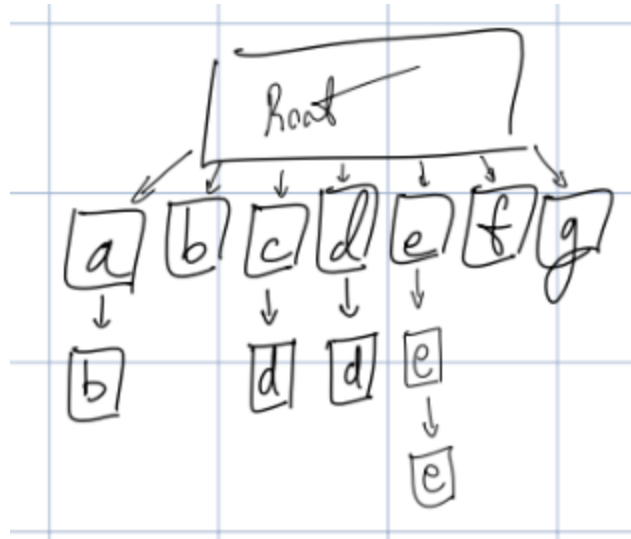
#### Step 4:

In step 4 we add “dd” to the struct. Since ‘d’ already exists in the first level, we will skip writing the first character and simply append the second character ‘d’ under the first level ‘d’. Once again, after a new addition to the tree, we jump back to the root



#### Step 5:

Similar to step 4, we will append ‘e’ to the first level ‘e’, then jump back to the root.



#### Step 6:

Similar to the previous steps, since, “e”->”e” already exists, we will skip over these two when attempting to add “eee”. We will then add an ‘e’ to the second level ‘e’. Once again, when we jump back to the root, as we had previously added a new word.

#### Step 7:

When attempting to add the last “ee”, we start from the root. However, because an ‘e’ already exists in the first level, we can jump to the next level. Additionally, we can remove one of the characters we need to append since it already exists. Thus we only need to append an ‘e’. Once again, we can see how it already exists, so we do not need to append anything. Thus, we are done!

Thus, in short, the input text was split up such that it was added in this particular order: “abcdefg”, “ab”, “cd”, “dd”, “ee”, “eee”, “ee”. It should once again be noted that this only occurred since a particular set of circumstances occurred. Should more letters be added that already exist within the tree, the tree will most likely become taller, rather than wider - which is better for file compression, as it means that there is more space saved in the compression of said file.

**Note: The reason the characters were split the way they were was for simplicity of understanding. Every time we jump back to the root, we don’t actually parse the text. This was only done in this example to demonstrate the steps.**

---

## 7. What I learned & Steps taken to complete assignment

From this assignment I learned how to implement data compression through the use of the Lempel-Ziv Compression. In this assignment, we use Lempel-Ziv Compression 1978, otherwise known as LZ78 for its publication in the year 1978. It is a lossless data compression algorithm that utilizes a dictionary to help store characters, from which symbols can be associated with every character. Though it could be possible to use a pre-initialized dictionary to speed up the process of compression, it would take up more space than necessary. Thus, the dictionary is created during the encoding process (compressing) as well as the decoding process (decompression).

On top of what I learned from the last assignment, I gained more experience on how to use pointers in C, as well as how to use structures. Especially from the last few assignments, I am now more comfortable with structures as well as how to access things within the structure. For example, one should use a `->` when attempting to access a pointer to a variable within a struct. It dereferences the pointer, allowing for access of the variable. One would use a `.` when accessing a member of a struct. From this assignment alone, I would say that I learned how to (or at least gained a somewhat decent understanding of) bit padding, usage of buffers (blocks), indexing within buffers, as well as read and writing headers for security.

From what I learned from this assignment, bit padding is when we add extra bits to ensure a standard size. In this assignment, the reason for this is (most probably) to ensure that symbols in a pair are always 8 bits in length, not only making it easier to read in but also limiting the maximum number it is able to create. What this means is that we are able to allocate up to 256 symbols when compressing a file; which is exactly what we want, since `TrieNode's children` variable is only able to accept up to 256 items (limited by `ALPHABET`, which is set to 256 in `trie.h`). What padding helps us to do is ensure that symbols will always have the bit length 8. For example, because the integer 13 is 4, we need to make sure it takes up 1 byte of space, thus we add extra zeros to extend the bit length of this symbol. It should be noted that since codes are not limited to 8 bits, padding is not required for them. Second thing that I learned from this assignment was the usage of buffers as well as indexing buffers. In this assignment we use buffers to store binary pairs as well as characters. Buffers were used in this assignment to assist in the compression and decompression of files. On top of the usage of buffers, I was also to index the buffers in order to keep track of the bits, and symbols in each respective buffer. For example, in the function `"write_pair"`, the function is to use a buffer to write a pair to an outfile. In this case a pair consists of a code and symbol. The bits of code are buffered first, then the symbol - both of which are expected to be buffered starting from LSB. In short, a buffer was used as a sort of 'staging ground' in which the file is restructured before being compressed. Another thing I learned in this assignment was reading and writing headers for security. In particular the `FileHeader` structure containing the magic number. From my understanding of it, it seems as if we use the magic number in the header to ensure that *decode* will only decompress items that were compressed by *encode*. Additionally, with the usage of protection in the header, we are able

to ensure that certain permissions such as file creation, reading of files, and writing of files (similar to the last assignment).

An error I had the hardest time with within `io.c` was the fact that my `read_pairs` was not functioning as expected. For example, an error that occurred when testing the file with the provided test from resources was that when the tester expected 3, my function returned 630. Eventually, I figured out that it was simply because I was not setting `*code` and `*sym` to zero at the beginning of the function. What this did was edit the previously set `*code` and `*sym` and unnecessarily shift random bits to the left that were either not meant to exist or didn't exist at all. It also (obviously) resulted in the returning completely incorrect results. On top of this, I also face an error where the test expected 176, but my function returned 2. I could not figure it out until it was announced that it was because my `read_pair` was not properly set up and it was reading even though there was nothing left to read, as it expected a `STOP_CODE` before it ever reached the end of file.

What I found interesting while implementing the Lempel-Ziv data compression was how simple it sounded in theory. Of course it was in theory and it quickly became a challenge upon the first few attempts to implement it. What helped me overcome this was discussing the assignment with TA Michael during his sections. He helped me understand at least the basic idea of the assignment as well as clarify my confusion on certain things within the assignment. I completed this assignment by working on all the files and functions more or less in the same order as presented in the assignment documentation. The first file I completed was `trie.c`, then `word.c`, `io.c`, `encode.c`, and lastly `decode.c`. I found `io.c` to be the most challenging in terms of debugging amongst all files, since there were so many functions that called upon one another. Additionally, there were various places in which something could go wrong, such as wrong indexing or incorrect bit manipulation. By wrong indexing what I mean is that the number of variables that were created and used to properly index the input and output buffers. On top of this, the errors I faced in flushing the remaining words and pairs.

---

## 8. How the efficiency of compression changes with entropy

The efficiency of compression changes with entropy because (usually) the higher the entropy, the lower the ratio of compression. In essence, the more random the characters are within the compression file, the lower the amount of compression will be. This is because the higher the entropy (randomness of characters in the file to be compressed), the Trie structure (dictionary) will usually be wider since there is less likely to be repeated words. The lower the entropy, the higher the rate of compression. This will be shown in the next example.

In this example, when we compress an infile consisting of 100 'a's, Because the data solely consists of a single character. What this does within our program is appending a bunch of the character 'a' to the Trie struct.

```
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaa
```

**Figure 5**

This figure shows the input file. Exactly 100 'a's are expected to be compressed. These characters will be compressed exactly the same way as was shown in a previous example demonstrating how the compression worked.

```
Compressed file size: 40 bytes
Uncompressed file size: 110 bytes
Compression ratio: 63.64%
```

**Figure 6**

This figure shows the size of the compressed file, uncompressed file, as well as the compression ratio. We are able to notice that the compression ratio is 63%. As stated before, the lower the entropy, the higher the rate of compression.

```
00000000  ac ba ad ba b4 81 00 00  85 19 b6 30 86 45 a1 61  |.....0.E.a|
00000010  a7 70 61 a5 90 61 a4 b0  61 a3 d0 61 42 e1 85 99  |.pa..a..aB...|
00000020  b0 a7 10 15 a2 02 00 00                |.....|
00000028
```

**Figure 7**

This figure shows the hexdump of the compressed file of figure 5.

```
fbsgfghage
eugibhgvsb
hinyjstorb
iroshgtshb
greivbesui
etrfjavbas
asvnjnsvbn
a0Isrg8eua
erugaeahvb
wae8uoarvh
```

**Figure 8**

This figure shows the input file that is to be compressed. Note how it consists of various random characters in no particular order. What this means is that there is a higher entropy; and what this in turn means is that the Trie used to compress this text file will be wider, rather than taller, as the characters in this file are (a) not in any pattern, and (b) if there is a pattern, there are only so many that would allow for compression.

```
Compressed file size: 114 bytes
Uncompressed file size: 110 bytes
Compression ratio: -3.64%
```

**Figure 9**

This figure shows the size of the compressed file, uncompressed file, as well as the compression ratio. We are able to immediately notice that the compression ratio is -3.64%. But how? The uncompressed file size is literally the same as the previous one! The reason we have a negative compression ratio is because of the high entropy, as could be seen in figure 8. The less patterns there are (higher entropy) in a file that is to be compressed, the less the algorithm is able to compress. As stated before, the lower the entropy, the higher the rate of compression.

```
00000000  ac ba ad ba b4 81 00 00 99 25 96 b9 9c e9 2c 68 | .....%....,h|
00000010  11 56 65 a1 10 65 51 57 69 83 56 76 44 4c a1 85 | .Ve..eQWi.VvDL..|
00000020  b4 e0 16 f2 82 1a a1 0b 6f 41 ee 28 48 b9 3b 77 | .....oA.(H.;w|
00000030  ce 02 1d 41 1b 0a 85 dc 92 16 d8 0d 65 44 9d a4 | ...A.....eD..|
00000040  b0 d0 61 66 55 98 28 86 cc 29 61 84 dd a4 36 cd | ..afU.(..)a...6.|
00000050  a5 6e eb 53 90 44 c8 15 38 4b 1d a2 b0 c8 31 67 | .n.S.D..8K....1g|
00000060  48 19 82 96 2a 04 77 38 0e f3 86 c8 89 68 01 05 | H...*.w8....h..|
00000070  00 00                                     | ..|
00000072
```

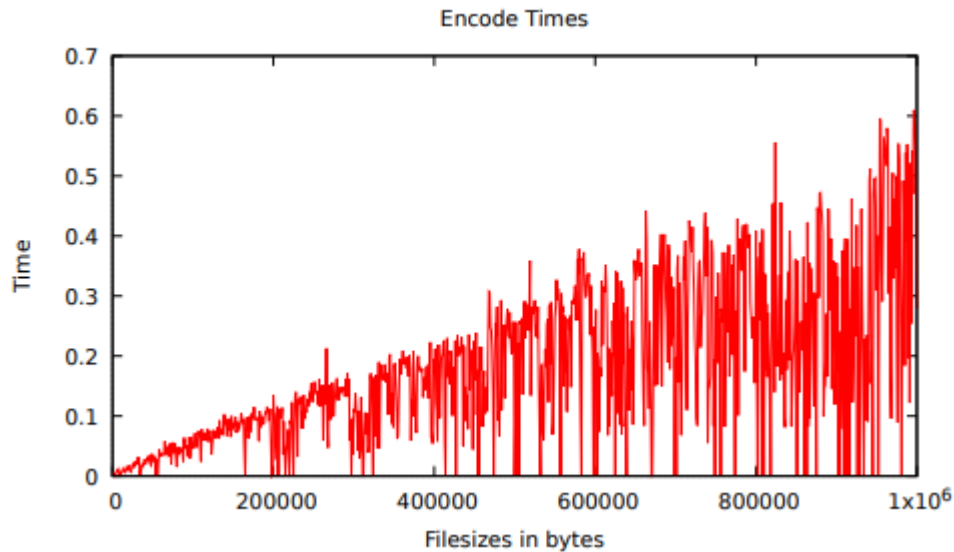
**Figure 10**

This figure shows the hexdump of the compressed file of figure 8

---

## 9. Graphs

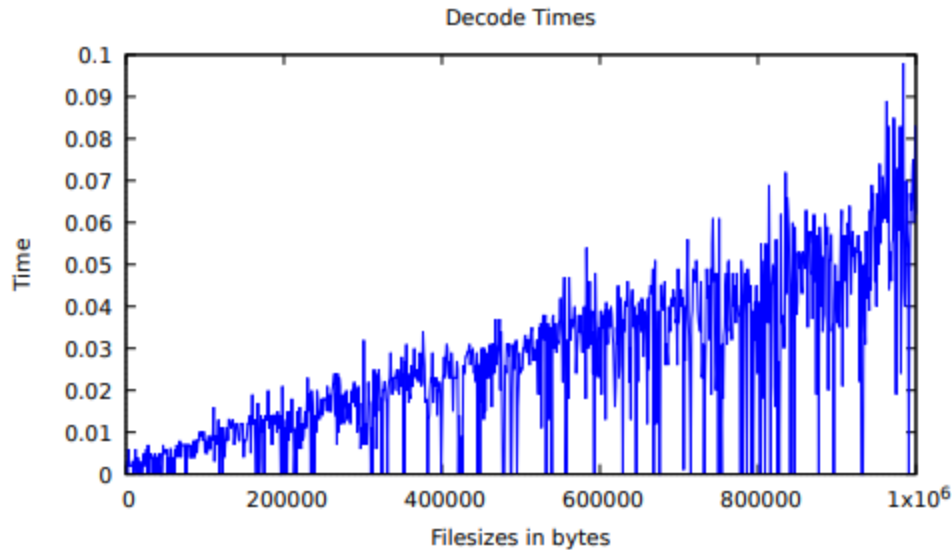
Though not entirely related to any particular part of this writeup, I thought it would be interesting to add some graphs relating to the time taken to compress and decompress files.



**Figure 11**

This figure depicts the time in seconds taken to compress files ranging from 0 to 1,000,000 bytes. What we are able to see is a general increase in the time taken to compress files with higher file sizes. This is usually because; (a) larger files have a higher chance of having higher entropy, (b) larger files have more characters to add to the algorithm's generated dictionary (trie in our case), taking up more time. In all honesty, Though it is completely understandable that there are fluctuations in time required for different file sizes, I do not have a concrete hypothesis as to why some file sizes take 0 seconds to compress. My best guess is that the randomizer used to create a randomized input file size created an input test file that had low entropy, allowing for a low compression time (almost zero). However, there could be other explanations for this.





**Figure 12**

This figure depicts the time in seconds taken to decompress files ranging from 0 to 1,000,000 bytes. What we see is a general increase in time taken to decompress larger file sizes, which is understandable, as there are (usually) more characters to build from the algorithm's dictionary.

Similar to what was stated for the previous figure, it is completely understandable for fluctuations to exist in time required for compression and decompression, however once again I have no concrete evidence as to why some file sizes take zero seconds to decompress. My best guess is exactly the same as stated in the previous figure description.

---

## 10.Credit:

**TA Michael Xiong - Helped me understand how LZ78 worked and what some of the functions were meant to be doing in this assignment. Basically clarified any confusion I had about this assignment.**