

# Assignment 7

## Lempel-Ziv Compression

Prof. Darrell Long  
CSE 13S – Winter 2021

Due: March 14<sup>th</sup> at 11:59 pm PST

*Everyday, we create 2.5 quintillion bytes of data – so much that 90% of the data in the world today has been created in the last two years alone.*

---

—IBM report on Big Data (2011)

### 1 Introduction

Compressing data means reducing the number of bits needed to represent it. Compressed data, due to its reduced size, is a lot faster to transfer and less expensive to store, freeing up storage and increasing network capacity. Algorithms that perform data compression are known as data compression algorithms. Data compression algorithms are divided into two categories: lossy and lossless. Lossy compression algorithms compress more than lossless compression algorithms, but at the cost of losing some information, which can't be recovered. Lossy compression algorithms are typically used for audio and video files, where a loss in quality is tolerable and often not noticeable. Lossless compression algorithms on the other hand don't compress as much, but do not lose any data, meaning compressed information can be exactly reconstructed back into its uncompressed form. Lossless compression algorithms are used for data that must maintain integrity, such as binaries, text documents, and source code.

### 2 Lempel-Ziv Compression

Abraham Lempel and Jacob Ziv published papers for two lossless compression algorithms, LZ77 and LZ78, published in 1977 and 1978, respectively. The core idea behind both of these compression algorithms is to represent repeated patterns in data with using pairs which are each comprised of a code and a symbol. A code is an unsigned 16-bit integer and a symbol is a 8-bit ASCII character.

We will explain this algorithm by working through an example. Assume we have some string “*abab*”. Assume we also have a dictionary where the key is a `prefix`, also known as a word, and the value is a code that we can utilize for fast look-ups. Since codes are 16-bit unsigned integers, the dictionary is limited to  $2^{16} - 1$  possible codes. Why don't we have  $2^{16}$  codes? Because we will be reserving a code to be used as a stop code which indicates the end of our encoded data. We will define the maximal usable code as `MAX_CODE`, which has the value  $2^{16} - 1$ . The dictionary is initialized with the empty word, or a string of zero length, at the index `EMPTY_CODE`, which is a macro for 1. We will specify the first index in which a new word can be added to as the macro `START_CODE`, which has the value 2.

We now consider the first character: ‘*a*’. Since this is the first character, we know that the string of all previously seen characters up to this point must be the empty word; we haven't considered any characters prior to this point. We then append the current character ‘*a*’ to the empty word, which yields the word “*a*”. We check the dictionary in vain to see if we've encountered this word before. Since we haven't yet seen this word, we will add this first word

## Compression Example

Initialized dictionary		Adding “a”		Adding “b”		Adding “ab”	
Word	Code	Word	Code	Word	Code	Word	Code
“”	EMPTY_CODE	“”	EMPTY_CODE	“”	EMPTY_CODE	“”	EMPTY_CODE
	START_CODE	“a”	START_CODE	“a”	START_CODE	“a”	START_CODE
	3		3	“b”	3	“b”	3
	4		4		4	“ab”	4
	...		...		...		...
	MAX_CODE		MAX_CODE		MAX_CODE		MAX_CODE

to the dictionary and assign it the first available code of 2, or START\_CODE. We set our previously seen word to be the empty word and output the pair (EMPTY\_CODE, ‘a’).

We continue on, now considering the second character: ‘b’. We append this character to our previously seen word, which yields “b” as our current word. We again check the dictionary in vain to see if we’ve encountered this word before. Since we haven’t, we add this word to the dictionary and assign it the next available code of 3. We set the previously seen word to the empty word and output the pair (EMPTY\_CODE, ‘b’).

The next character we see is ‘a’. We append this character to our previously seen word, which yields “a” as our current word. We check the dictionary to see if we’ve encountered this word before. Indeed we have. We set the previously seen word to “a” and proceed to the next character without any further output.

We read in the last character, ‘b’. We append this character to our previously seen word, which yields “ab”. Clearly, this word isn’t in the dictionary, so pair comprised of the current symbol, ‘b’, and the code of the previously we add it and assign it the next available code of 4. What should we output? The seen word. Since our previously seen word at this stage was “a”, this code will be 2. Thus, we output (2, ‘b’). To finish off compression, we output the final pair (STOP\_CODE, 0). As you can imagine, this pair signals the end of compressed data. The symbol in this final pair isn’t used and the value is of no significance. The macro STOP\_CODE has the value of 0.

Of course, a compression algorithm is useless without a corresponding decompression algorithm. Assume we’re reading in the output from the preceding compression example. The output was comprised of the following pairs, in order: (EMPTY\_CODE, ‘a’), (EMPTY\_CODE, ‘b’), (2, ‘b’), (STOP\_CODE, 0). Similar to the compression algorithm, the decompression algorithm initializes a dictionary containing only the empty word. The catch is that the key and value for the decompressing dictionary is swapped; each key is instead a code and each value a word. As you might imagine, the decompression algorithm is the inverse of the compression algorithm, and thus from the output pairs, the decompression algorithm will recreate the same dictionary used during compression to output the decompressed data.

We consider the first pair: (EMPTY\_CODE, ‘a’). We append the symbol ‘a’ to the word denoted by EMPTY\_CODE, which is the empty word. Thus, the current word is “a”. We add this word to the dictionary and assign it the next available code of 2, then output the current word “a”.

We consider the next pair: (EMPTY\_CODE, ‘b’). We append the symbol ‘b’ to the word denoted by EMPTY\_CODE, which is the empty word. Thus, the current word is “b”. We add this word to the dictionary and assign it the next available code of 3, then output the current word “b”.

We now consider the next pair: (2, ‘b’). We append the symbol ‘b’ to the word denoted by the code 2, which we previously added to our dictionary. The word denoted by this code is “a”, whence we obtain our current word of “ab”. We add this word to the dictionary and assign it the next available code of 4, then output the current word “ab”. Finally, we read in the last pair: (STOP\_CODE, 0). Since the code is STOP\_CODE, we know that we have finished decompression.

If the basic idea behind the compression and decompression algorithms do not immediately make sense to you,

### Decompression Example

Initialized dictionary		Adding “a”		Adding “b”		Adding “ab”	
Code	Word	Code	Word	Code	Word	Code	Word
EMPTY_CODE	“”	EMPTY_CODE	“”	EMPTY_CODE	“”	EMPTY_CODE	“”
START_CODE		START_CODE	“a”	START_CODE	“a”	START_CODE	“a”
3		3		3	“b”	3	“b”
4		4		4		4	“ab”
...		...		...		...	
MAX_CODE		MAX_CODE		MAX_CODE		MAX_CODE	

or if you desire a more visual representation of how the algorithms work, make sure to attend section and get help early! **Things will not get easier as time goes on.**

## 3 Your Task

Your task is to implement two programs called `encode` and `decode` which perform LZ78 compression and decompression, respectively. The requirements for your programs are as follows:

1. `encode` can compress any file, text or binary.
2. `decode` can decompress any file, text or binary, that was compressed with `encode`.
3. Both operate on both little and big endian systems. *Interoperability is required.*
4. Both use variable bit-length codes.
5. Both perform read and writes in efficient blocks of 4KB.
6. Both `encode` and `decode` must interoperate with the provided binaries – not just your code.

## 4 Specifics

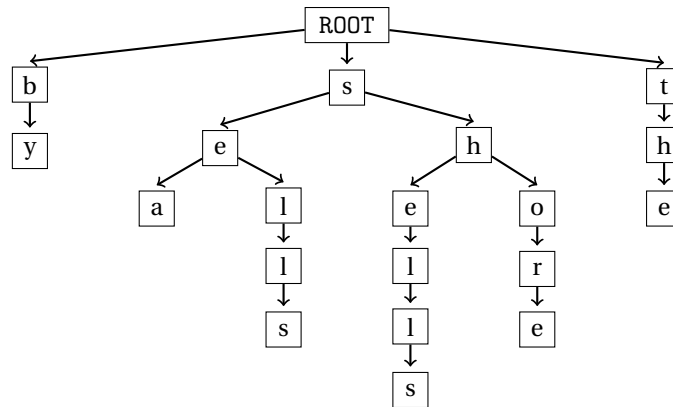
You will need to implement some new ADTs for this assignment: an ADT for tries and an ADT for words. In addition to these new ADTs, you will need to be concerned about variable-length codes, I/O, and endianness.

### 4.1 Tries

The most costly part of compression is checking for existing prefixes, or words. You could utilize a hash table, or just an array to store words, but that wouldn't be optimal, as many of the words you need to store are prefixes of other words. Instead you choose to utilize a *trie*.

A trie<sup>1</sup> is an efficient information re-*trie*-val data structure, commonly known as a prefix tree. Each node in a trie represents a symbol, or a character, and contains  $n$  child nodes, where  $n$  is the size of the alphabet you are using. In most cases, the alphabet used is the set of ASCII characters, so  $n = 256$ . You will use a trie during compression to store words.

<sup>1</sup>Edward Fredkin, “Trie memory.” *Communications of the ACM* 3, no. 9 (1960): 490–499.



Above is an example of a trie containing the following words: “*She*”, “*sells*”, “*sea*”, “*shells*”, “*by*”, “*the*”, “*sea*”, “*shore*”. Searching for a word in the trie means stepping down for each symbol in the word, starting from the root. Stepping down the trie is simply checking if the current node we have traversed to has a child node representing the symbol we are looking for, and setting the current node to be the child node if it does exist. Thus, to find “*sea*”, we would start from the trie’s root and step down to ‘*s*’, then ‘*e*’, then ‘*a*’. If any symbol is missing, or the end of the trie is reached without fully matching a word, while stepping through the trie, then the word is not in the trie. You *must* follow the specification shown below when implementing your trie ADT.

#### 4.1.1 TrieNode

```

1 struct TrieNode {
2     TrieNode *children[ALPHABET];
3     uint16_t code;
4 };

```

The `TrieNode` struct will have the two fields shown above. Each trie node has an array of 256 pointers to trie nodes as children, one for each ASCII character. It should be easy to see how this simplifies searching for the next character in a word in the trie. The code field stores the 16-bit code for the word that ends with the trie node containing the code. This means that the code for some word “*abc*” would be contained in the trie node for ‘*c*’. Note that there isn’t a field that indicates what character a trie node represents. This is because the trie node’s index in its parent’s array of child nodes indicates what character it represents. The `trie_step()` function will be repeatedly called to check if a word exists in the trie. A word only exists if the trie node returned by the last step corresponding to the last character in the word isn’t `NULL`.

#### 4.1.2 TrieNode \*trie\_node\_create(uint16\_t code)

Constructor for a `TrieNode`. The node’s code is set to `code`. Make sure each of the children node pointers are `NULL`.

#### 4.1.3 void trie\_node\_delete(TrieNode \*n)

Destructor for a `TrieNode`. Note that only a *single* pointer is passed here. The destructors you have written in the past have taken double pointers in order to `NULL` the pointer by dereferencing it. For this assignment, a single pointer is much more manageable and simplifies the code you have to otherwise write.

#### 4.1.4 TrieNode \*trie\_create(void)

Initializes a trie: a root TrieNode with the code EMPTY\_CODE. Returns the root, a TrieNode \*, if successful, NULL otherwise.

#### 4.1.5 void trie\_reset(TrieNode \*root)

Resets a trie to just the root TrieNode. Since we are working with finite codes, eventually we will arrive at the end of the available codes (MAX\_CODE). At that point, we must reset the trie by deleting its children so that we can continue compressing/decompressing the file. Make sure that each of the root's children nodes are NULL.

#### 4.1.6 void trie\_delete(TrieNode \*n)

Deletes a sub-trie starting from the trie rooted at node n. This will require recursive calls on each of n's children. Make sure to set the pointer to the children nodes to NULL after you free them with trie\_node\_delete().

#### 4.1.7 TrieNode \*trie\_step(TrieNode \*n, uint8\_t sym)

Returns a pointer to the child node representing the symbol sym. If the symbol doesn't exist, NULL is returned.

## 4.2 Word Tables

Although compression can be performed using a trie, decompression still needs to use a look-up table for quick code to word translation. This look-up table will be defined as a new struct called a WordTable. Since we can only have  $2^{16} - 1$  codes, one of which is reserved as a stop code, we can use a fixed word table size of UINT16\_MAX, where UINT16\_MAX is a macro defined in inttypes.h as the maximum value of a unsigned 16-bit integer. Hint: this has the exact same value as MAX\_CODE, which we defined earlier.

Why aren't we using hash tables to store words? Because there is a *fixed* number of codes. Each index of this word table will be a new struct called a Word. You will store words in byte arrays, or arrays of uint8\_t. This is because strings in C are null-terminated, and problems with compression occur if a binary file is being compressed and contains null characters that are placed into strings. Since we need to know how long a word is, a Word will also have a field for storing the length of the byte array, since we can't use string.h functions like strlen() on byte arrays. You *must* use the following specification for the new Word ADT.

### 4.2.1 Word

```
1 struct Word {  
2     uint8_t *syms;  
3     uint32_t len;  
4 };
```

A Word holds an array of symbols, syms, stored as bytes in an array. The length of the array storing the symbols a Word represents is stored in len.

#### 4.2.2 WordTable

```
1 typedef Word * WordTable
```

To make things easier to reason about, we can define an array of Words as a WordTable.

#### 4.2.3 Word \*word\_create(uint8\_t \*syms, uint32\_t len)

Constructor for a word where syms is the array of symbols a Word represents. The length of the array of symbols is given by len. This function returns a Word \* if successful or NULL otherwise.

#### 4.2.4 Word \*word\_append\_sym(Word \*w, uint8\_t sym)

Constructs a new Word from the specified Word, w, appended with a symbol, sym. The Word specified to append to may be empty. If the above is the case, the new Word should contain only the symbol. Returns the new Word which represents the result of appending.

#### 4.2.5 void word\_delete(Word \*w)

Destructor for a Word, w. Like with trie\_node\_create() in §4.1.3, a single pointer is used here to reduce the complexity of memory management, thus reducing the chances of having memory-related errors.

#### 4.2.6 WordTable \*wt\_create(void)

Creates a new WordTable, which is an array of Words. A WordTable has a pre-defined size of MAX\_CODE, which has the value UINT16\_MAX. This is because codes are 16-bit integers. A WordTable is initialized with a single Word at index EMPTY\_CODE. This Word represents the empty word, a string of length of zero.

#### 4.2.7 void wt\_reset(WordTable \*wt)

Resets a WordTable, wt, to contain just the empty Word. Make sure all the other words in the table are NULL.

### 4.3 Codes

It is a requirement that your programs, encode and decode, be able to read and write pairs containing variable bit-length codes. As an example, assume we are going to write the pair (13, 'a'). What is the minimum number of bits needed to represent 13? We can easily calculate the minimum number of bits needed to represent any integer  $x \geq 1$  using the formula  $\lfloor \log_2(x) \rfloor + 1$ . The only edge case is with 0, whose bit-length is 1. Thus, we calculate that the minimum number of bits needed to represent 13, or the bit-length of 13, to be 4. That being said, in practice, the bit-length of the code in the pair you're going output wouldn't be the bit-length of the code itself, but rather the bit-length of the *next available code* assignable, as described in the earlier compression example. The reason for this is because decompression *must* know at all times the bit-length of the code it is going to read. Because the dictionary constructed by decompression contains exactly the words and codes contained by the trie constructed by compression, it is evident that, by using the bit-length of the next available code, that compression and decompression will agree on the variable bit-lengths of codes. Note that only codes have variable bit-lengths: symbols in a pair are always 8 bits.

As an example, assume we are going to output the pair (13, 'a'), and the next available code assignable in our dictionary is 64. We will need to convert this pair to binary, starting with the code. As stated earlier, the bit-length of the code is the bit-length of the next available code. We calculate that the bit-length of 64 is 7. We start from the least significant bit of our code, 13, and construct the code portion of the pair's binary representation: LSB

1011000 MSB. Note that the LSB is on the left and the MSB is on the right. Zeroes are padded because the bit-length of 64 is 7, while the bit-length of 13 is 4, which is why there are three padded zeroes. In the same fashion, we start from the least significant bit of our symbol, 'a', and add the symbol portion of the pair's binary representation to the previous code portion, which yields LSB 101100010000110 MSB, since the ASCII value of 'a' is 97.

Now assume that we are reading in the binary LSB 101100010000110 MSB and converting that back into a pair. We know that the next available code assignable by compression and decompression are the same at every step. Thus, we know that compression must have output the pair's code with the bit-length of 64, which is 7. We go through the first 7 bits of the binary: 1, 0, 1, 1, 0, 0, and 0, summing up the bits as we go, simulating how positional numbering systems work. This means we perform  $1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 + 0 \cdot 2^4 + 0 \cdot 2^5 + 0 \cdot 2^6 = 13$ , exactly the code output by compression. We do the same for the remaining 8 bits to reconstruct the symbol.

#### code.h

```
1 #ifndef __CODE_H__
2 #define __CODE_H__
3
4 #include <inttypes.h>
5
6 #define STOP_CODE 0 // Signals end of decoding/decoding.
7 #define EMPTY_CODE 1 // Code denoting the empty Word.
8 #define START_CODE 2 // Starting code of new Words.
9 #define MAX_CODE UINT16_MAX // Maximum code.
10
11 #endif
```

## 4.4 I/O

It is also a requirement that your programs, encode and decode, perform efficient I/O. Reads and writes will be done 4KB, or a block, at a time, which implicitly requires that you buffer I/O. Buffering is the act of storing data into a buffer, which you can think of as an array of bytes. You will be implementing an I/O module for this assignment, and its API is explained in the following sections.

### 4.4.1 FileHeader

```
1 struct FileHeader {
2     uint32_t magic;
3     uint16_t protection;
4 };
```

This is the struct definition for the file header, which contains the magic number for your program and the protection bit mask for the original file. The file header is the first thing that appears in a compressed file. The magic number field, `magic`, serves as a unique identifier for files compressed by `encode`. `decode` should only be able to decompress files which have the correct magic number. This magic number is 0xBAADBAAC.

Before writing the file header to the compressed file using `write_header()`, you must swap the endianness of the fields if necessary since *interoperability is required*. If your program is run on a system using big endian, the fields

must be swapped to little endian, since little endian is canonical. A module specifically for handling endianness will be provided.

#### 4.4.2 `int read_bytes(int infile, uint8_t *buf, int to_read)`

This will be a useful helper function to perform reads. As you may know, the `read()` syscall *does not* always guarantee that it will read all the bytes specified. For example, a call could be issued to read a block of bytes, but it might only read half a block. So, we write a wrapper function to *loop calls* to `read()` until we have either read all the bytes that were specified (`to_read`), or there are no more bytes to read. The number of bytes that were read are returned. **You should use this function whenever you need to perform a read.**

#### 4.4.3 `int write_bytes(int outfile, uint8_t *buf, int to_write)`

This function is very much the same as `read_bytes()`, except that it is for looping calls to `write()`. As you may imagine, `write()` isn't guaranteed to write out all the specified bytes (`to_write`), and so we loop until we have either written out all the bytes specified, or no bytes were written. The number of bytes written out is returned. **You should use this function whenever you need to perform a write.**

#### 4.4.4 `void read_header(int infile, FileHeader *header)`

This reads in `sizeof(FileHeader)` bytes from the input file. These bytes are read into the supplied header. Endianness is swapped if byte order isn't little endian. Along with reading the header, it must verify the magic number.

#### 4.4.5 `void write_header(int outfile, FileHeader *header)`

Writes `sizeof(FileHeader)` bytes to the output file. These bytes are from the supplied header. Endianness is swapped if byte order isn't little endian.

#### 4.4.6 `bool read_sym(int infile, uint8_t *sym)`

An index keeps track of the currently read symbol in the buffer. Once all symbols are processed, another block is read. If less than a block is read, the end of the buffer is updated. Returns true if there are symbols to be read, false otherwise.

#### 4.4.7 `void write_pair(int outfile, uint16_t code, uint8_t sym, int bitlen)`

“Writes” a pair to outfile. In reality, the pair is buffered. A pair is comprised of a code and a symbol. The bits of the code are buffered first, starting from the LSB. The bits of the symbol are buffered next, also starting from the LSB. The code buffered has a bit-length of `bitlen`. The buffer is written out whenever it is filled.

#### 4.4.8 `void flush_pairs(int outfile)`

Writes out any remaining pairs of symbols and codes to the output file.

#### 4.4.9 `bool read_pair(int infile, uint16_t *code, uint8_t *sym, int bitlen)`

“Reads” a pair (code and symbol) from the input file. The “read” code is placed in the pointer to code (e.g. `*code = val`) The “read” symbol is placed in the pointer to sym (e.g. `*sym = val`). In reality, a block of pairs is read into a buffer. An index keeps track of the current bit in the buffer. Once all bits have been processed, another block is read. The first `bitlen` bits are the code, starting from the LSB. The last 8 bits of the pair are the symbol, starting from the LSB. Returns true if there are pairs left to read in the buffer, else false. There are pairs left to read if the read code is not `STOP_CODE`.



**4.4.10** `void write_word(int outfile, Word *w)`

“Writes” a pair to the output file. Each symbol of the Word is placed into a buffer. The buffer is written out when it is filled.

**4.4.11** `void flush_words(int outfile)`

Writes out any remaining symbols in the buffer to the outfile.

Note that the output file in which you write to must have the same protection bits as the original file. Like in assignment 4, you will make use of `fstat()` and `fchmod()`.

All reads and writes in this program must be done using the system calls `read()` and `write()`, which means that you must use the system calls `open()` and `close()` to get your file descriptors. As stated earlier, all reads and writes must be performed in efficient blocks of 4KB. **You will want to use two static 4KB `uint8_t` arrays to serve as buffers: one to store binary pairs and the other to store characters.** Each of these buffers should have an index, or a variable, to keep track of the current byte or bit that has been processed.

## 5 Program Options

Your encode program must support the following `getopt()` options:

- `-v` : Print compression statistics to `stderr`.
- `-i <input>` : Specify input to compress (`stdin` by default)
- `-o <output>` : Specify output of compressed input (`stdout` by default)

Your decode program must support the following `getopt()` options:

- `-v` : Print decompression statistics to `stderr`.
- `-i <input>` : Specify input to decompress (`stdin` by default)
- `-o <output>` : Specify output of decompressed input (`stdout` by default)

The verbose option enables a flag to print out informative statistics about the compression or decompression that is performed. These statistics include the compressed file size, the uncompressed file size, and *space saving*. The formula for calculating space saving is:

$$100 \times \left( 1 - \frac{\text{compressed size}}{\text{uncompressed size}} \right).$$

The verbose output of both encode and decode must match the following:

Compressed file size: X bytes  
Uncompressed file size: X bytes  
Space saving: XX.XX%

## 6 Compression

The following steps for compression will refer to the input file descriptor to compress as `infile` and the compressed output file descriptor as `outfile`.

1. Open `infile` with `open()`. If an error occurs, print a helpful message and exit with a status code indicating that an error occurred. `infile` should be `stdin` if an input file wasn't specified.

2. The first thing in `outfile` must be the file header, as defined in the file `io.h`. The magic number in the header must be `0xBAADBAAC`. The file size and the protection bit mask you will obtain using `fstat()`. See the man page on it for details.
3. Open `outfile` using `open()`. The permissions for `outfile` should match the protection bits as set in your file header. Any errors with opening `outfile` should be handled like with `infile`. `outfile` should be `stdout` if an output file wasn't specified.
4. Write the filled out file header to `outfile` using `write_header()`. This means writing out the `struct` itself to the file, as described in the comment block of the function.
5. Create a trie. The trie initially has no children and consists solely of the root. The code stored by this root trie node should be `EMPTY_CODE` to denote the empty word. You will need to make a copy of the root node and use the copy to step through the trie to check for existing prefixes. This root node copy will be referred to as `curr_node`. The reason a copy is needed is that you will eventually need to reset whatever trie node you've stepped to back to the top of the trie, so using a copy lets you use the root node as a base to return to.
6. You will need a monotonic counter to keep track of the next available code. This counter should start at `START_CODE`, as defined in the supplied `code.h` file. The counter should be a `uint16_t` since the codes used are unsigned 16-bit integers. This will be referred to as `next_code`.
7. You will also need two variables to keep track of the previous trie node and previously read symbol. We will refer to these as `prev_node` and `prev_sym`, respectively.
8. Use `read_sym()` in a loop to read in all the symbols from `infile`. Your loop should break when `read_sym()` returns false. For each symbol read in, call it `curr_sym`, perform the following:
  - (a) Set `next_node` to be `trie_step(curr_node, curr_sym)`, stepping down from the current node to the currently read symbol.
  - (b) If `next_node` is not `NULL`, that means we have seen the current prefix. Set `prev_node` to be `curr_node` and then `curr_node` to be `next_node`.
  - (c) Else, since `next_node` is `NULL`, we know we have not encountered the current prefix. We write the pair `(curr_node->code, curr_sym)`, where the bit-length of the written code is the bit-length of `next_code`. We now add the current prefix to the trie. Let `curr_node->children[curr_sym]` be a new trie node whose code is `next_code`. Reset `curr_node` to point at the root of the trie and increment the value of `next_code`.
  - (d) Check if `next_code` is equal to `MAX_CODE`. If it is, use `trie_reset()` to reset the trie to just having the root node. This reset is necessary since we have a finite number of codes.
  - (e) Update `prev_sym` to be `curr_sym`.
9. After processing all the characters in `infile`, check if `curr_node` points to the root trie node. If it does not, it means we were still matching a prefix. Write the pair `(prev_node->code, prev_sym)`. The bit-length of the code written should be the bit-length of `next_code`. Make sure to increment `next_code` and that it stays within the limit of `MAX_CODE`. Hint: use the modulo operator.
10. Write the pair `(STOP_CODE, 0)` to signal the end of compressed output. Again, the bit-length of code written should be the bit-length of `next_code`.
11. Make sure to use `flush_pairs()` to flush any unwritten, buffered pairs. Remember, calls to `write_pair()` end up buffering them under the hood. So, we have to remember to flush the contents of our buffer.
12. Use `close()` to close `infile` and `outfile`.

## 7 Decompression

The following steps for decompression will refer to the input file to decompress as `infile` and the uncompressed output file as `outfile`.

1. Open `infile` with `open()`. If an error occurs, print a helpful message and exit with a status code indicating that an error occurred. `infile` should be `stdin` if an input file wasn't specified.
2. Read in the file header with `read_header()`, which also verifies the magic number. If the magic number is verified then decompression is good to go and you now have a header which contains the original protection bit mask.
3. Open `outfile` using `open()`. The permissions for `outfile` should match the protection bits as set in your file header that you just read. Any errors with opening `outfile` should be handled like with `infile`. `outfile` should be `stdout` if an output file wasn't specified.
4. Create a new word table with `wt_create()` and make sure each of its entries are set to `NULL`. Initialize the table to have just the empty word, a word of length 0, at the index `EMPTY_CODE`. We will refer to this table as `table`.
5. You will need two `uint16_t` to keep track of the current code and next code. These will be referred to as `curr_code` and `next_code`, respectively. `next_code` should be initialized as `START_CODE` and functions exactly the same as the monotonic counter used during compression, which was also called `next_code`.
6. Use `read_pair()` in a loop to read all the pairs from `infile`. We will refer to the code and symbol from each read pair as `curr_code` and `curr_sym`, respectively. The bit-length of the code to read is the bit-length of `next_code`. The loop breaks when the code read is `STOP_CODE`. For each read pair, perform the following:
  - (a) As seen in the decompression example, we will need to append the read symbol with the word denoted by the read code and add the result to `table` at the index `next_code`. The word denoted by the read code is stored in `table[curr_code]`. We will append `table[curr_code]` and `curr_sym` using `word_append_sym()`.
  - (b) Write the word that we just constructed and added to the table with `write_word()`. This word should have been stored in `table[next_code]`.
  - (c) Increment `next_code` and check if it equals `MAX_CODE`. If it has, reset the table using `wt_reset()` and set `next_code` to be `START_CODE`. This mimics the resetting of the trie during compression.
7. Flush any buffered words using `flush_words()`. Like with `write_pair()`, `write_word()` buffers words under the hood, so we have to remember to flush the contents of our buffer.
8. Close `infile` and `outfile` with `close()`.

## 8 Deliverables

You will need to turn in:

1. Makefile:
  - `CFLAGS=-Wall -Wextra -Werror -Wpedantic` must be included.
  - `CC=clang` must be specified.
  - `make clean` must remove all files that are compiler generated.
  - `make encode` should build your encode program.
  - `make decode` should build your decode program.

- `make` should build both `encode` and `decode`, as should `make all`.
  - Your programs should have no memory leaks.
2. Your programs *must* have the following source and header files:
    - `encode.c`: contains the `main()` function for the `encode` program.
    - `decode.c`: 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. *You must not modify this file.*
    - `word.c`: the source file for the Word ADT.
    - `word.h`: the header file for the Word ADT. *You must not modify this file.*
    - `io.c`: the source file for the I/O module.
    - `io.h`: the header file for the I/O module. *You must not modify this file.*
    - `endian.h`: the header file for the endianness module. *You must not modify this file.*
    - `code.h`: the header file containing macros for reserved codes. *You must not modify this file.*
  3. You may have other source and header files, but *do not try to be overly clever*.
  4. Running `scan-build` yields no bugs. If there are any false positives, make sure to explain why they are false positives in your `README.md`.
  5. `README.md`: This must be in Markdown. This should describe how to use your program and Makefile. This also contains any explanations for complaints generated by `scan-build`.
  6. `DESIGN.pdf`: This must be a PDF. The design document should describe your design for your program with enough detail that a sufficiently knowledgeable programmer would be able to replicate your implementation. This does not mean copying your entire program in verbatim. You should instead describe how your program works with supporting pseudo-code.
  7. Working `encode` and `decode` programs, along with test files, and other source files will be supplied in the resources repository.

## 9 Submission

To submit your assignment, refer back to `assignment0` for the steps on how to submit your assignment through `git`. Remember: *add*, *commit*, and *push*!

Your assignment is turned in *only* after you have pushed. If you forget to push, you have not turned in your assignment and you will get a *zero*. “I forgot to push” is not a valid excuse. It is *highly* recommended to commit and push your changes *often*.

## A LZ78 Algorithm Pseudocode

### A.1 Compression

COMPRESS(*infile*, *outfile*)

```
1  root = TRIE_CREATE()
2  curr_node = root
3  prev_node = NULL
4  curr_sym = 0
5  prev_sym = 0
6  next_code = START_CODE
7  while READ_SYM(infile, &curr_sym) is TRUE
8      next_node = TRIE_STEP(curr_node, curr_sym)
9      if next_node is not NULL
10         prev_node = curr_node
11         curr_node = next_node
12     else
13         WRITE_PAIR(outfile, curr_node.code, curr_sym, BIT-LENGTH(next_code))
14         curr_node.children[curr_sym] = TRIE_NODE_CREATE(next_code)
15         curr_node = root
16         next_code = next_code + 1
17     if next_code is MAX_CODE
18         TRIE_RESET(root)
19         curr_node = root
20         next_code = START_CODE
21     prev_sym = curr_sym
22 if curr_node is not root
23     WRITE_PAIR(outfile, prev_node.code, prev_sym, BIT-LENGTH(next_code))
24     next_code = (next_code + 1) % MAX_CODE
25 WRITE_PAIR(outfile, STOP_CODE, 0, BIT-LENGTH(next_code))
26 FLUSH_PAIRS(outfile)
```

### A.2 Decompression

DECOMPRESS(*infile*, *outfile*)

```
1  table = WT_CREATE()
2  curr_sym = 0
3  curr_code = 0
4  next_code = START_CODE
5  while READ_PAIR(infile, &curr_code, &curr_sym, BIT-LENGTH(next_code)) is TRUE
6      table[next_code] = WORD_APPEND_SYM(table[curr_code], curr_sym)
7      WRITE_WORD(outfile, table[next_code])
8      next_code = next_code + 1
9      if next_code is MAX_CODE
10         WT_RESET(table)
11         next_code = START_CODE
12 FLUSH_WORDS(outfile)
```