LZW compression, named after its three founders, Abraham Lempel, Jacob Ziv, and Terry Welch showcases a modern means of shrinking a file, despite being discovered shortly after the personal computer. Despite its rather long existence, it still impacts contemporary technology through the GIF, TIFF, and PDF file formats. Its simple and straightforward implementation, along with its lossless format, has assisted in its success throughout the years.

First, LZW compression relies heavily on an underlying symbol table. The symbol table’s implementation is insignificant, but the quickest access and insert times possible are highly recommended. This table is initialized with all 256 (28 because of 8-bits) ASCII characters; these will be the initial “substrings” for the current uncompressed file. Then, a program loops through the file. For every iteration, the beginning of the file is scanned for the longest codeword contained within the codebook, or symbol table; this is called the longest prefix. So, for example, if the file began as “a little boy once was […]”, with the initialized symbol table, the longest prefix in this case would be “a”. Then, that prefix, concatenated with its neighbor character to the right, are placed into the symbol table with its appropriate index; in this case, that new codeword is “a “ with the codeword value 257 because a space is the next character in line. Depending on the language chosen to implement LZW, the file, or ‘input’ string, is shortened and moved past the encoded codeword (in this situation, “a”). Now, the input string reads “ little boy once was […]”. One critical step in this compression is actually writing out the compressed data to a file. Because the codeword found in this example is “a”, which happens to be the value 97 in ASCII (the initialized codebook), that value is written to a new file in binary with N-bits. In a fixed-length encoding, where N is the number of encoded bits, the codebook size never exceeds 2N, and every codeword written to the compressed file is N-bits. However, in a variable-length encoding, one may start with 8-bit codewords and move up to 16-bits, allowing for 216 (65,536) codewords. Due to its lossless quality, expansion returns the file back to its original contents; this is key in files such as PDFs as mentioned earlier. When expanding, read in N bits at a time (expansion must be fixed-length or variable-length based on what was used in compression), adding codewords in a similar manner as compression. Going back to the compression example, if the value 97 was read in, then go to the value in the symbol table associated with 97 (“a”), write it to the uncompressed file, and append that string to the last string read in. In the first iteration of expansion, the empty string will be concatenated (since nothing had been read in previously). But, in the example with “a”, in the second iteration, the program would read in “ “, write it to the uncompressed file, then append it to “a” (“a “) and insert it into the codebook. Now, that code, or longest prefix, can be used later on in the expansion of the file. In this project, these basic principles will be expanded to support a few more forms of compression and expansion.

For example, variable-length encoding was mentioned earlier. Currently, LZW.java uses fixed-length encoding of 12-bit codewords; to improve on this, MyLZW.java will begin with 8-bits and progress to 16-bit codewords. Variable-length encoding showcases its usefulness in files with common substrings. For example, in the English language, there are words that appear often (“the”, “and”, “him”, etc…), which map neatly to a codebook and are bound to appear multiple times throughout a document. However, when a file contains a significant amount of random noise, with no patterns whatsoever, LZW is hard pressed to find common longest prefixes. This proposed change will be tested and examined on its practical utility. The final couple of changes to LZW.java are a reset and monitor mode. In reset mode, when the codebook fills up (contains 216 codewords), then reset the entire codebook. This helps by allowing new codewords to be introduced as one iterates through the file, potentially compressing the file even further, but it clears out the already-added codewords, which may remove some of the most common words in the file, worsening compression. For monitor mode, compress normally, but when the codebook fills up, monitor the compression ratios of the file. Take the old compression ratio, which was calculated before the symbol table filled up, and divide it by a new compression ratio, which is calculated for every iteration while the table is full. If this result is greater than a specific threshold, in other words the old compression method would result in a significantly larger file than just resetting the codebook, reset the codebook. These three proposed changes are simple, but have the potential to offer dramatic improvements to LZW.java.

Implementing these modifications, despite rather tiny in length of code, is complex. First step, start small and aim for the variable-length encoding, then advance to completing the modes. When coding variable-length encoding, the number of bits output to a file changes as the algorithm continues. Because the number of bits changes, so does the size of the codebook. So, the first immediate change is ridding of the ‘final’ keywords in LZW.java for both the variables W (bit-length) and L (size of codebook); they now dynamically