**LZW Compression**

Just like any compression algorithm, the LZW compression algorithm has a compression step and an expansion step. The compression step takes a string and represents the data in a way which takes up less space on disk, but the computer cannot read. It stores a codebook (a symbol table) which allows the expansion step to read the information and represent it back in a way that can be natively read by the computer. In each step, the codebook is initialized as every ASCII character. The compression step scans the input until there are no unread characters. It finds the longest string in the codebook which is a prefix of the remaining unread characters, writes the codeword associated with the string to the file, and adds the next character in the string to the codeword and saves it in the codebook for future use. The compressed file will include the codebook representation as well. For expansion, the algorithm reads the compressed file for the first codeword. Then, it will look up the codeword in the codebook and output the characters corresponding to it. Finally, in order to keep up with the codebook, the previous pattern is added to the first character of the next codeword and added to the book.

In this example, the codewords in the book are of a fixed width, meaning that both a single character and 5 characters are represented by the same number of bits in the compressed file. Because of this, LZW compression as implemented in the book is a slow starting compression algorithm. Meaning that until the codebook starts to fill up, no compression is actually achieved. In fact, we are actually losing space to start off. For example, a single character can be represented with 8 bits in extended ASCII. If our algorithm is set to read 12-bit long codewords, reading a single character would mean *adding* 4 bits to the compressed file. It is not until we begin to fill up the codebook that we start to see actually compression. To solve this, we could use variable-width codewords. The idea of variable width codewords is to decrease the time that it takes for us to see actual compression. If we start with codewords of 9 bits, we are only wasting 1 bit by reading a single character instead of 4. Variable width codewords also allows us to increase the size of our codebook to further increase compression. In this project, the maximum we will go to is 16 bits, which allows for 216 = 65,536 codewords instead of 4096 with a static 12-bit codeword width.

In both the compression and expansion section of the algorithm, there is a variable which can be used to determine the current state of the codebook. I will use this to decide exactly when to increment the width of the code which will be read. First, however, I need to modify the final variables at the top of the class in order to start at a 9-bit length codeword and a codebook with a length of 512. The variables I need to modify to achieve this are W and L respectively. Then, in the compression step I will check to see when the code variable equals the length of the codebook. When this occurs, the next code that is added will fill up the codebook. However, instead of allowing it to fill up, I will increment the width variable and double the length of the codebook. This will cause the next codes to be written to the compressed file at width 10. This will occur up until the width is equal to 16 bits. Once the width hits 16 bits the width will no longer be incremented, and the codebook will be allowed to fill up. The next step will depend on the user’s choice and will be discussed later.

I will take a similar approach in the expansion portion. First, I will adjust the codebook to be an array of length 65536 since this is the maximum amount of codes which can be stored after expanding the width to 16. I am comfortable initializing an array of this size because the amount of memory it takes up is miniscule despite the large amount of space. After the algorithm adds the prefix to the codebook, I will check to see if the i variable – the variable which holds the current spot in the codebook – is equal to the length of the codebook. Even though the array is fully initialized to the maximum amount of spaces at the beginning, I am still going to keep track of an arbitrary length variable to determine the right time to expand the code width. Also, the width is expanded when it is 1 less than the length because if you wait until it is equal, the expansion will be a step ahead of the compression and read the wrong number of bits. Similar to before, when this condition is true I will increment the width and double the length. This will not occur when the width is already 16, and the codebook will fill up.

In order to handle the codebook fill ups, we first must handle the extra command line argument which will be required so the user can make a choice. The three choices that we have when the codebook fills are to continue with the full codebook, reset the codebook and start from the beginning, and to monitor the compression ratio and then reset when it reaches a certain threshold. The command line arguments which correspond to these will be ‘n’, ‘r’, and ‘m’ respectively. This only matters when the user wants to compress. If the user chooses to compress, I will check if there is a 2nd command line argument. If there is not, or the argument exists but is not one of the choices, an IllegalArgumentException will be thrown. Otherwise, I will pass an integer to the compression function corresponding to the choice – 1 for ‘n’, 2 for ‘r’, and 3 for ‘m’. In the compression function, after the symbol table has been initialized to every ASCII value, the integer will be written as the very first 2 bits of the output file. This is essential to communicate which method was used when compressing to the expansion function. In the expansion function after initializing the symbol table and before beginning the actual expansion, I will read the first 2 bits of the file and save the corresponding character in an integer variable for use later.

Implementing the “do nothing” method in MyLZW is simple because that is the default behavior for the LZW algorithm as originally implemented. For the other two methods, special instructions will be carried out to implement them. However, since “doing nothing” is the natural behavior of this LZA implementation, I will simply let the program run as usual if the argument is neither ‘r’ or ‘m’.

Implementing the “reset” method requires a few steps. The code for resetting the codebook will only run if the user entered ‘r’ into the arguments when compressing the file. First, I will need to determine when the codebook is filled. In order to do this in the compression function, the code variable needs to be monitored. The code variable holds the next index in which a code will be inserted in the codebook. This was already done for the implementation of variable width codewords. However, the codebook is only filled up if the width is 16. Otherwise, the codebook can still be expanded. Once width is 16 and the next index is equal to the length of the codebook at width 16 (65536), then the codebook is considered full. To completely reset, the codebook must be set to the state which it was at when it was first initialized – with only the 256 ASCII characters in the TST. Then, the code, width, and length variables need to be reset to their original states. They will need to be R+1, 9, and 512 respectively. This needs to be done to ensure that the correct number of bits are being written, and the next code is added to the correct position in the codebook.

The expansion function reset will have similar functionality to the compression reset but requires another step. The expansion function will run with codebook reset when the bits corresponding to the argument present during compression is 2. During expansion, the codebook is full when i is equal to length of the codebook and width is equal to 16. When this occurs, the next value that is inserted will overflow the codebook and it needs to be reset. The symbol table will be reset to the original state – a length 65536 array with indices 0 through 255 set to the standard ASCII values – and the i, width, and length variables will need to be set back to their original values similar to before during compression. Checking whether or not the codebook is full in the expansion step has to come before the codeword is updated instead of after. This is because when the first value is read in the expansion step, nothing is added to the codebook. This ensures that the two codebooks stay in sync when being built.

The monitor portion only requires a few extra steps, and they are the same for compression and expansion. In order to keep track of the compression ratio, I will make two variables: one to hold the number of bits being read from the input file, and one to hold the number of bits being written to the output file. These will represent the uncompressed vs compressed bits for compression, and the compressed vs uncompressed bits for expansion. Then, if the user has chosen to enter monitor mode, the ratio will be calculated the first time the codebook gets filled. This first ratio will be saved for comparison later. Then, every time a new value is read, another ratio will be calculated from the uncompressed and compressed bits. The beginning ratio will be divided by the current ratio. If this final ratio is greater than 1.1, then the codebook will reset exactly like before. If the ratio is less than 1.1, we will continue monitoring the ratio until the file ends or the ratio increases past the threshold.

Overall, each portion of the implementation has a main idea which is central to making the feature work. For variable length codewords, the most important part is comparing the codebook position to the length of the codebook and increasing the width and length when appropriate. For the implementation of reset, it is important to completely reset the symbol table to its original state. I will use methods outside of the expansion and compression function to do this, since it can be called from wherever in the file. The width and length also need to be reset to their original values, as well as the position of the codebook reset to the number of the ASCII values. For implementation of monitor, it is integral to keep track of the number of bits that have been compressed from the uncompressed bits. Then, create the ratios from those values so we know when the codebook should be reset. Lastly, since the classes used to handle reading and writing use a lot of calls to java’s substring method – which has runtime of O(n) – the compression speed will be bound by these calls. It is possible to change this for faster compression but is not necessary to achieve compression.