When compressing a file, the optimal situation is that you achieve the smallest file size in the compressed version of the file. In the original LZWmod file given, there is a slight amount of compression achieved, but it is not anywhere close to as significant as the modified versions of LZWmod. For example, the ratio of compression for the original LZWmod was 1.1955 for the bmps.tar file. This means that we essentially achieved almost no compression, being that the compressed file size was only *slightly* smaller than the original file. However, using the modified version of LZWmod and the “n” keyword, signifying that no reset of the dictionary would occur, the ratio improved *dramatically* to 13.6682. This means that whenever the codeword bits weren’t set to a fixed number, and were able to expand from 9 – 16, the compression was able to achieve a much higher ratio. The highest compression was achieved from the extra credit portion I did, which reset the dictionary automatically when the size of the read bits divided by the size of the compressed bits (the ratio size) was over 2.3. The ratio achieved was 14.0400. Of all the tests, I found the large.txt compression test to be the most interesting. Using the original LZWmod, the ratio was 20171. Using the do-nothing command, the ratio was 2.4328. Using the reset command, the ratio actually got *worse,* and was 2.3144*.* And finally, the extra credit LZW\_mod achieved the same ratio as the reset. To speculate these results, I opened the large.txt text file open, to reveal what appeared to be the text of some play. Knowing that an LZW algorithm stores codewords based off of size and whether the codeword was seen before or not, I came to the conclusion that the compression got worse when resetting the dictionary because all of the previous words that were stored were deleted, and there were most likely a lot of repeat words. This means that this algorithm will have to re-add all of these, therefor making the compression file size larger in the end. I believe that the “do nothing” mode resulted in a higher compression ratio due to the fact that it had variable sized codeword bit lengths, and with the diversity of words inside of this file it was able to store more words in the dictionary and store them to larger lengths so when the repeated words came it was able to compress them more easily. Perhaps another interesting test was the frosty.jpg, due to it being an image file. For all 4 of the LZW tests, *all* of them proved to have **worse** than a 1.0 compression rate, rendering the entire concept of LZW compression pointless. I even attempted to change the compression ratio reset on the extra credit higher than normal, and lower than normal. Both of these yielded results that the compression file size was larger than the original file, and I believe this to be because of how images get changed to characters and special characters inside of this algorithm, rendering the pattern aspect to prove not so prominent. Whenever using the Mac OSX compression, the results were surprisingly close to my implementation of LZW. For example, using the “n” keyword for large.txt, my algorithm actually achieved a better compression ratio. However, for bmps.tar and frosty.jpg, the Mac OSX compression had better results, although not by much.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Original LZW compression | Ratio | Modified LZW “n” compression | Ratio | Modified LZW “r” compression | Ratio | Extra Credit compression | Ratio |
| Large.txt | 605,184 | 2.0171 | 501,776 | 2.4328 | 527,449 | 2.3144 | 527,449 | 2.3144 |
| Bmps.tar | 925,079 | 1.1955 | 80,912 | 13.6682 | 80,912 | 13.6682 | 78,769 | 14.0400 |
| Frosty.jpg | 177,453 | 0.7143 | 163,788 | 0.7739 | 171,169 | 0.7405 | 163,788 | 0.7739 |

|  |  |  |
| --- | --- | --- |
|  | MAC OSX Compression | Ratio |
| Large.txt | 493,577 | 2.4318 |
| Bmps.tar | 64,186 | 17.2299 |
| Frosty.jpg | 127,254 | 0.9960 |