In general, the unix compress LZW algorithm gave the most consistently performant results and Sedgewick’s LZW.java performed the worst, especially in test cases which punished its lack of flexibility most. In most cases, my implementation of a more adaptable LZW approach was comparable to the compress command, particularly when prompted to reset the dictionary when all the 16-bit codewords had been exhausted.

All four algorithms gave poor results for “frosty.jpg” and “Lego-big.gif.” The compress utility simply stated that it was impossible to compress the files further, whereas all three Java programs substantially inflated the file sizes. This isn’t terribly surprising because jpg and gif are already compressed file formats. In fact, the gif file is already compressed with LZW! The entropy of the files is simply too great to meaningfully compress further. Even bzip2 expands these files slightly. Other than the gif and jpg files, the executable file was the most entropic. Sedgewick’s code expanded this file, though the other three algorithms achieved similar modest compression ratios. The biggest factor here then must be the variable-length codewords.

The bitmap files started to expose some of the problem with Sedgewick’s implementation. For “wacky.bmp,” compress and LZWmod.java with and without dictionary reset performed identically and heavily compressed the file. It was easy to compress this file because it was mostly white pixels. LZW.java also heavily compressed the file, but the 12-bit codewords from the outset caused it not to achieve quite as performant results as the other algorithms. The “gone\_fishing.bmp” got worse results from all parties because it had much less repeated data, but again LZW.java didn’t achieve as much early compression for the small file because it started out using a larger codeword size. Sedgewick’s code really got stung by the bmp archive and the windows NT logo files. In the case of the archive, the dictionary quickly filled on “gone\_fishing.bmp” and no longer had space to properly form long enough runs to exploit the long white pixel runs in “wacky.bmp.” It must have stumbled on the NT logo file because of its lack of variable codeword length alone because all three other files produced identical results without leveraging dictionary reset. They also had identical results for the bmp archive because they could zoom very efficiently through its largest file, “wacky.bmp,” without using very many codewords at all. We know the dictionary could not have reset in these cases because it would have caused my algorithm to have different results for the reset version if it had.

The text files had more varied results. The smallest three files were too small to cause a reset and thus gave equivalent answers for LZWmod.java and compress and slightly worse results for the author’s code. Interestingly, the large text file got noticeably different ratios from each algorithm. Surprisingly, the most performant algorithm was LZWmod.java *without* the dictionary reset. It must have been that both my implementation with the reset and the compress algorithm performed disadvantageous dictionary resets. The compress did a little better likely because it waited for performance degradation to dump the still-useful codewords, but it’s interesting that even after its intelligent threshold was triggered its decision ended up not paying off with additional compression here. When given the whole text archive, the same strategy ended up paying clear dividends for the compress algorithm, likely because it was able to adapt to the differences between the prose-filled files and the code-filled files at the correct times. As with all the archives, LZW.java did comparatively much worse here than on the individual files since it had lost its ability to adapt to new information relatively early in the compression run.

When all the file types were bundled together, the compress utility was the clear winner because it was the most intelligently adaptive to changes in the data. The resetting version of LZWmod.java did nearly as well, though the fact that its resetting was not carefully timed gave it slightly worse performance than compress. The two algorithms without the capacity to throw out the old codewords at and start afresh did substantially worse with this data. This is not that surprising. The data is long and variable. An adaptive algorithm which fails to adapt pays the price when things change.