**Modifying LZW: Obstacles and Solutions**

The modifications I made to Sedgewick’s code in LZW.java each addressed a shortcoming in his implementation. To make sure everything worked, I ran compress and expand back and forth for the tested files and verified that the bit counts matched the originals. To be extra sure the algorithms were synched correctly, I also ran the bash utility diff on the transformed files versus the originals. Since I implemented the solution piece by piece, I split the discussions of each into a problem/solution pair and considered them in the sequence I tackled them.

**PROBLEM 1: The input needs to be changed to stream a single byte/char at a time.**

**SOLUTION:** Use the author’s input method readChar() to get single bytes from the input file as needed. I adjusted the author’s method to return an end-of-file sentinel value rather than throw an exception when invoked with no more data to return. This made it always safe to call readChar() and easier to test for the end of file in my loop. It didn’t affect the author’s implementation in LZW.java because he doesn’t call this method at all. There’s a bigger problem, though.

**PROBLEM 2: We want to avoid storing the entire input file in a single String when encoding.**

**ATTEMPT:** Stream characters into a single StringBuilder object as we build prefixes. Overwrite used parts of the StringBuilder as the data is processed, keeping track of how long the current prefix is so that it is clear what parts of the StringBuilder contain valid data. This limits the memory overhead from the size of the uncompressed file to the size of the longest codeword formed. Search the dictionary each time a byte is added for the current prefix. When we find an invalid prefix, add the whole prefix (including the mismatch) to the dictionary. Output the encoding of the codeword *without* the last character which caused the mismatch to the file, then swap the mismatched character into position 0. Stop at the EOF sentinel after one final output. This worked okay, but I found a better way later.

**PROBLEM 3: The author uses a TST, which does not offer constant time lookup/insert.**

**ATTEMPT:** Use a different data structure, obviously. The likely candidates here are either a hash table or a trie of some kind, since both offer Ɵ(1) insert and search. I decided to implement the dictionary as a 256-way trie. I already had a similar implementation from my first assignment for this data structure that I could readily modify. Plus, this seemed like the simplest option for something to write that I was sure would work without much fuss. I just adjusted the nodes so that they could store values, set the branching to 256, and put in some simple methods to make the structure work for specified substrings of a StringBuilder. This was like what it already did for searchPrefix(StringBuilder s, int start, int end) from assignment 1 so it was easy to adjust. It worked, but there was a catch.

**OBSTACLE:** I timed the runs of the different implementations while I observed their compression performance because I wanted to examine the impact of my choice here. The author’s code clearly had performance issues for several reasons and ran in about 7 minutes on the biggest archive file as a point of reference here. It’s a little unfair to compare my program to the Unix compress utility, since Java programs are obviously going to run more slowly than their equivalent lower-level language (e.g. C) counterparts. Still, for some of the files I noticed that my algorithm performed significantly worse than I expected (i.e., took about 4 seconds instead of well below 1 second to run). It took longest for wacky.bmp and the archives which contain it.

Why did this happen? This file had lots and lots of repeated data. This, in turn, created very long codewords. My searches were scaling poorly with the increase in the length of the codeword. Specifically, it was hurting me to search for a long codeword character by character only to append just ONE more character before repeating almost the *exact same* search. For very long codewords, I was getting runs of performance which were quadratic in the length of the codeword. This meant that the best-case compression scenario was giving me my worst-case runtime!

**REVISED SOLUTION (TO # 2 & 3):** One small conceptual change to the trie simplified and improved my approach considerably. I added another instance reference to the trie which points to the most recently accessed node of the trie much like an iterator. I included a different method which accepts a char one at a time and simply adjusts this pointer down the trie structure and returns true if the given path exists, otherwise changes nothing and returns false. If that method returns false, I know to output the codeword where the iterator is currently referencing. Then, to add that codeword in to the dictionary, I simply need to put one more node in the chain at that character’s location in the iterator’s children array with the appropriate next codeword. The last step of adding to the dictionary is to reset the iterator pointer back to the root so we may begin forming another prefix from scratch.

First, this solved my StringBuilder problem in a much more elegant way than I’d solved it the first time. I no longer needed a StringBuilder at all. I could store a single character value in LZWmod and a single extra pointer reference in the trie. That was all the memory I needed to maintain the current state because each additional character **always** either branches from the most recent node or adds itself to the dictionary from the most recent node and then begins a new prefix as its first character. I could just stream in char by char and let the new and improved trie handle most of the state.

More importantly, this allowed my program to interface with the trie in constant time. This isn’t just constant time relative to the size of the dictionary. It’s constant time relative to the size of the dictionary **AND** relative to the length of the current codeword. In empirical terms, this change cut my runtime for the problem scenarios from ~4 seconds to something like .17 second. While this didn’t improve the compression at all, this is a huge improvement in runtime. It’s still not quite as fast as compress, but it no longer was embarrassingly slow for a system utility.

**PROBLEM 4: The number of bits per codeword in the author’s code is fixed at 12.**

**SOLUTION:** Implement a variable length encoding scheme which starts at 9-bit codewords and caps at 16-bit codewords. The first and most obvious adjustment here was to change the L and W identifiers in the author’s LZW.java file from constants to variables. I set the initial value of W to 9 since that’s where we want it to begin. The number of words we can represent in W bits is 2^W, so I also had to adjust L to 2^9, which is 512.

Whenever the current codeword is equal to the value of L, all the available codewords we can represent in W bits have been used. How do we fix it? So long as W is still less than 16, increment it. Then set L to the new maximum number of codewords. We could accomplish this by setting it to 2^W, but it’s cheaper to either multiply L by 2 or bit shift L left by 1. We can think of this either as doubling the possible range of codewords, or by moving the bits we’re using over by 1 so we have room for one more. We keep incrementing the size of the codewords in this fashion one at a time until W is equal to 16. I didn’t even need to change the way writing to the file works because it takes W as a parameter for how much data to write already. So long as we make the adjustment in the right place, it all works the same way as before (only better).

This works the same way whether we are performing the encode or the decode algorithm. The only difference is that the decode algorithm checks the condition earlier in its process because it’s a playing catch-up with the encode algorithm. It tests *before* reading whereas the encode algorithm tests *after* writing. What happens if L is 16? That brings us to the last problem.

**PROBLEM 5: The dictionary never resets its code when full.**

**SOLUTION:** Encode whether the dictionary should reset in a single bit flag at the start of an encoded file and let the user decide how to run the utility from the command line. Changing the I/O to get an option from the user for the utility is a trivial alteration. The first real task here was to figure out how to put a single bit in the beginning of the file so that it was recoverable during decompress in a way that didn’t mess up the information streaming. I read some of the Java I/O documentation and didn’t find anything terribly useful. I was getting a little worried that I was going to have to implement some unpleasant bitwise operations myself. Luckily, I decided to read through all the author’s binary I/O methods first. He had already implemented write(boolean x) and readBoolean() methods which solved all these concerns for me. I just had to write whatever the user input first during encode and read it back in first during decode.

Now encode and decode know whether they’re supposed to reset the dictionary. but how do they do it? This condition is tested only when all the codewords are used (code == L) and the codewords are already the maximum length (W == 16). I made the reset testing condition an else block after testing if (W < 16) for the variable bit encoding. Resetting the dictionary means minimally setting W back to 9, L back to 512, and the current next codeword to assign to 257 (since 256 is the author’s sentinel) for both compress and expand. We need to reset the symbol table indices after 256 in expand() to null because it relies on finding unused locations to handle its special case. We also need to reset the values back to just ASCII in compress so that we don’t use codewords expand can’t find. I wrote a simple method in MWTforLZW.java to do that rather than making a whole new object.