University of Pittsburgh

**CS 1501 – Algorithm Implementation**
**Sherif Khattab**

# Assignment#3 - LZW[1]

**Due: Friday November 6th @ 11:59pm**

**Late submission: Sunday November 8th @11:59pm with 10% penalty per late day**

## OVERVIEW

**Purpose:** The purpose of this assignment is to fully understand the LZW compression algorithm, its performance and its implementation. We will improve the performance of the textbook's LZW implementation when the input files are large.

**Goal 1:** Read the input file as a stream of bytes (i.e., byte by byte) instead of all at once (feel free to use your lab code for this task).

**Goal 2:** Avoid the Theta(n) overhead of using String.substring() (feel free to use your lab code for this task).

**Goal 3:** Allow the codebook size to increase beyond the 4096 entries in the textbook's implementation using adaptive codeword width.

**Goal 4:** Allow LZW to learn new patterns after the codebook size is reached by giving the user the option to reset the codebook.

## PROCEDURE

1) Thoroughly read the description/explanation of the LZW compression algorithm as discussed in lecture and in the Sedgewick text.
2) Read over and make sure you understand the code provided by Sedgewick in file LZW.java in the handouts section on Canvas.
3) There are three fundamental problems with this code as given:
    a. The code reads the entire file as a single string, then compresses the characters in this string. This is problematic in that a very large file could not effectively be fit into a single Java string. Further, since JDK 7, the substring() method in the Java String class actually generates a new String object, causing a lot of overhead when used in the manner shown in the author's LZW.java file. In fact, with the textbook code using JDK 7+ and a large input file, the compression can take an excessive amount of time (ex: hours). This is due primarily to the code on line 30 of the LZW.java file:

[1] Assignment adapted from Dr. John Ramirez’s CS 1501 class.

```
input = input.substring(t);
```

The purpose of this statement is to shift down in the string past the characters that have already matched in the LZW dictionary. However, since in JDK 7+ the substring() method generates a new String, the effect of this statement is to create a copy of the entire file string, minus a few characters at the beginning that already matched in the dictionary. It is easy to see how this process repeated many many times (thousands and even millions) can slow the program execution incredibly.

b. The code uses a fixed length, relatively small codeword size (12 bits). With this limit, the program will run out of codewords relatively quickly and will not handle large files (especially archives) well.

c. When all code words are used, the program continues to use the existing dictionary for the remainder of the compression. This may be ok if the rest of the file to be compressed is similar to what has already been compressed, but it may not be.

4) In this assignment you will modify the author's code so as to correct (somewhat) these three problems. Proceed in the following way:

   a. Download the implementation of the algorithm provided and get it to work. Follow the instructions in the comments and run the program on a few test files to get familiar with using it. Try running the program with a large input file to see the behavior discussed above.

   b. Examine the code very carefully, convincing yourself exactly what is accomplished by each function and by each statement within each function.

   c. Copy the code to a new file called LZWmod.java and modify the code so that during compress() the input file is read as a stream of characters (bytes) rather than as a single string. There are many ways to do this and most of the details are up to you. However, here are a few requirements:

      i. The input file must be read in a single character (byte) at a time. Look over some classes in java.io that might be appropriate to use in this case. You may also utilize any of the author's IO classes if you wish. The idea is that rather than using the longestPrefixOf() method on a single String, you will find the longest prefix yourself by repeatedly reading and appending characters and looking the prefix up in the symbol table.

      ii. The "strings" that are looked up in the dictionary must actually be StringBuilder objects. Using StringBuilder rather than String will allow the values to be updated more efficiently (ex: appending a character onto the end of the StringBuilder will not require a new StringBuilder object to be created, as it would with a String).

      iii. The dictionary in which the StringBuilders are looked up must be some type of symbol table where the keys are StringBuilder objects and the values are arbitrary Java types. The dictionary must have (average) constant lookup time. Note that the predefined Java Hashtable and HashMap classes are not appropriate in this case because StringBuilder does not override the hashCode() method to actually hash the string value. Some options that could work include modifying one of the author's hash classes, the TST class, or the author's TrieST class so that they work for StringBuilder. Another option is to modify your DLB class from Assignment 1 to allow it to store the codewords.

   Note that you should not have to modify the expand() code for this feature at all. Also note that this modification will take just a few lines of code but it may take a lot of trial and error before you get it working properly. I strongly recommend getting this to work before moving on to parts d) and e) below. However, if you cannot get this to work, you can still get credit for parts d) and e) below by performing it on the original LZW.java file.

d. Also modify the code so that the LZW algorithm has a varying number of bits, as discussed in lecture. Your codeword size should vary from 9 bits to 16 bits and should increment the bitcount when all codes for the previous size have been used. This also does not require a lot of modification to the program, but you must REALLY understand exactly what the program is doing at each step in order to do this successfully (so you can keep the compress and decompress processes in sync). Once you get the program to work, thoroughly test it to make sure it is correct. If the algorithm is correct, the byte count of the original file and the uncompressed copy should be identical. Some hints about the variable-length code word implementation are given later on in this assignment.

e. As a partial solution to the issue of the dictionary filling, you will give the user the option to reset the dictionary or not via a command line argument. See more details on the command line arguments below, but the argument "r" will cause the dictionary to reset once all (16-bit) code words have been used.

    i. You will have to add code to reset the dictionary to the LZWmod.java file. As discussed in lecture this option will erase and reset the entire dictionary and start rebuilding it from scratch. As with the variable bits, be careful to sync both the compress and decompress to reset the dictionary at the same point.

    ii. Since now a file may be compressed with or without resetting the dictionary, in order to decompress correctly your program must be able to discern this fact. This can be done quite simply by writing a flag or sentinel at the beginning of the output file (actually only 1 bit is needed for this). Then, before decompression, your program will read this flag and determine whether or not to reset the dictionary when running out of codewords.

5) The author's interface already has a command-line argument to choose compression or decompression. File input and output can be supplied using the standard redirect operators for standard I/O: Use "<" to redirect the input to be a file and use ">" to redirect the output to be a file. Modify the interface so that for compression a command-line argument also allows the user to choose how to act when all codewords have been used. This extra argument should be an "n" for "do nothing", or "r" for "reset". Note that these arguments are only used during compression – for decompression the algorithm should be able to automatically detect which technique was used and decompress accordingly.

For example, assuming your program is called LZWmod.java, if you wish to compress the file bogus.txt into the file bogus.lzw, resetting the dictionary when you run out of codewords, you would enter at the prompt:

```
$ java LZWmod - r < bogus.txt > bogus.lzw
```

To prevent headaches (especially during debugging), you should not replace the original file with the new one (i.e., leave the original file unchanged). Thus, make sure you use a name for the output file that is different from the input file. If you then want to decompress the bogus.lzw file, you might enter at the prompt

```
$ java LZWmod + < bogus.lzw > bogus2.txt
```

The file bogus2.txt should now be identical to the file bogus.txt. Note that there is no flag for what to do when the dictionary fills – this should be obtained from the front of the compressed file itself (which, again, requires only a single bit).

6) Once you have your LZWmod.java program working, you should analyze its performance. A number of files to use for testing are provided on Canvas. Specifically, you will compare the performance of 4 different implementations:
   a. The original LZW.java program using codewords of 12 bits (i.e. the way it is originally – you don't have to change anything)
   b. Your modified LZWmod.java program with the streaming input text and variable length BITS from 9 to 16 as explained above, **without** dictionary reset.
   c. Your modified LZWmod.java program with the streaming input text and variable length BITS from 9 to 16 as explained above, **with** dictionary reset.
   d. The predefined Unix compress program (which also uses the lzw algorithm). If you have a Mac or Linux machine you can run this version directly on your computer. If you have a Windows machine, you can download this version of compress.exe (obtained originally from http://ww.willus.com/archive/ ). To decompress with this program use the flag "-d".

   Run all programs on all of the files and for each file record the original size, compressed size, and compression ratio (original size / compressed size).
   [Note: Because of the aforementioned run-time issues with the author's original code, it may take a prohibitive amount of time to get results for the larger files. However, it should eventually complete – just leave yourself a lot of time for your runs.

7) Write a short (~2 page) paper that discusses each of the following:
   a. How all four of the lzw variation programs compared to each other (via their compression ratios) for each of the different files. Where there was a difference between them, be sure to explain (or speculate) why. To support your assertions, include a table showing all of the results of your tests (original sizes, compressed sizes and compression ratios for each algorithm).
   b. For all algorithms, indicate which of the test files gave the best and worst compression ratios, and speculate as to why this was the case. If any files did not compress at all or compressed very poorly (or even expanded), speculate as to why.

8) **Important Notes:**

   a. In the author's code the bits per codeword (W) and number of codewords (L) values are constants. However, in your version you will need them to be variables. Clearly, as the bits per codeword value increases, so does the number of code words value.
   b. The symbol table that you use for the compression dictionary (ex: Trie, DLB) can grow dynamically, so you do not have to alter this as the codeword size increases. However, for the expand() method an array of String is used for the dictionary. Make sure this is large enough to accommodate the maximum possible number of code words.

c. Carefully trace what your code is doing as you modify it. You only have to write a few lines of code for this program, but it could still require a substantial amount of time to get to work properly. Clearly, the trickiest parts occur when the bits per codeword values are increased and when the dictionary is reset. It is vital that these changes be made in a consistent way during both compress and decompress. I recommend tracing these portions of code, either on paper or with output statements to make sure your compress and expand sections are treating them correctly. One idea is to have an extra output file for each of the compress() and expand() methods to output any trace code. Printing out (codeword, string) pairs in the iterations just before and after a bit change or reset is done can help you a lot to synchronize your code properly.

## EXTRA CREDIT

If you want to try some extra credit on this assignment, you can implement the reset in seamless way, so that the user does not have to specify whether or not to reset the dictionary. As discussed in lecture, this would involve some type of monitoring of the compression ratio once the codewords are all used and a reset would occur only when the compression ratio degrades to some level (you may have to do some trial and error to find a good value for the reset trigger level).

## SUBMISSION REQUIREMENTS

You must submit to **Gradescope** at least the following files:

1) Your LZWmod.java file

2) The modified Symbol Table implementation (e.g., TSTmod.java)

3) Any other files that you have written

4) Assignment Information Sheet.

5) The 2-page writeup

The idea from your submission is that your TA can compile and run your programs **from the command line** WITHOUT ANY additional files or changes, so be sure to test it thoroughly before submitting it. If the TA cannot compile or run your submitted code it will be graded as if the program does not work.

If you cannot get the programs working as given, clearly indicate any changes you made and clearly indicate why on your Assignment Information Sheet. You will lose some credit for not getting it to work properly, but getting the main programs to work with modifications is better than not getting them to work at all. A template for the Assignment Information Sheet can be found in the assignment's Canvas folder. You do not have to use this template but your sheet should contain the same information.

**Note: If you use an IDE such as NetBeans, Eclipse, or IntelliJ, to develop your programs, make sure they will compile and run on the command line before submitting – this may require some modifications to your program (such as removing some package information).**

**Note: Do not** submit any of the test files (input or output) – this will waste an incredible amount of space on the submission site!

## RUBRICS

| | |
|---|---|
| **Program Design / Execution:** | |
| Char by char (byte by byte) input is correct: | 14 points |
| StringBuilder used for prefix searches: | 8 points |
| Symbol table implemented / modified correctly: | 8 points |
| Variable-bit LZW kind of works: | 8 points |
| Variable-bit LZW mostly works: | 8 points |
| Variable-bit LZW is completely correct: | 8 points |
| Reset option works correctly: | 12 points |
| **Results / Write-Up :** | |
| Results shown/correct for all tested files: | 8 points |
| Original compared to modified version in write-up: | 8 points |
| Comp. ratios of different files compared/explained: | 8 points |
| Documentation: | 5 points |
| Submission / Assignment Information Sheet: | 5 points |
| Extra Credit | 10 points |