ITCS 6114/8114: Algorithms and Data Structures
Programming Project 1: LZW Compression

The project is due **before midnight on Wednesday, March 16.** It should be submitted on Moodle.

Introduction

The Lempel–Ziv–Welch (LZW) algorithm is a lossless data compression algorithm. LZW is an adaptive compression algorithm that does not assume prior knowledge of the input data distribution. This algorithm works well when the input data is sufficiently large and there is redundancy in the data. Two examples of commonly used file formats that use LZW compression are the GIF image format served from websites and the TIFF image format. LZW compression is also suitable for compressing text files, and is the algorithm in the *compress* Unix file compression utility.

This algorithm has two steps:
1. Encoding/Compressing
2. Decoding/Decompressing

The interesting thing is that the encoding table, or dictionary, computed during the encoding process does not need to be explicitly transmitted. It can be regenerated from the coded/compressed data.

Encoding/Compressing

The LZW algorithm reads an input sequence of symbols, groups the symbols into strings, and represents the strings with integer codes. Since the codes require less space than the strings, compression is achieved. LZW is a greedy algorithm in that it finds the longest string that it has a code for, and then outputs that code. LZW starts out with a known dictionary of single characters that constitute the input character set. For example, this could be the extended ASCII set of 256 characters (for the case of 8 bits) that represent alphabets, numbers, punctuation symbols, etc., and it uses these as the "standard" character set (see www.asciitable.com). It then reads symbols (i.e., characters) 8 bits at a time (e.g., 't', 'h', 'e',etc.) and appends them one by one to the current string. Each time it appends a symbol to the string, it checks whether the resulting string has a code in the dictionary. If it does, it reads in the next symbol and appends it to the current string. If the resulting string does not exist in the dictionary, it means it has found a new string: it outputs the code corresponding to the string without the newest symbol, adds the string concatenated with the newest symbol (i.e., the new string) to the dictionary with its code (which is the previous largest code value incremented by one), and resets the current string to the newest symbol. Thus the next time the LZW algorithm encounters a repeated string in the input sequence, it will be encoded with a single number. The algorithm continues to process symbols from the input sequence, building new strings until the sequence is exhausted, and it then outputs the code for the remaining string. Usually a bit length for the codes is specified, based on the maximum number of entries for the dictionary, so that the process does not use a very large amount of memory. So when the codes representing the strings are 12 bits long, the table size is $2^{12} = 4096$. It is necessary for the bit length of the codes to be longer than that of the characters (12 bits vs. 8 bits), but since repeated strings in the input sequence will be replaced by a single code, compression is achieved. LZW adaptively builds the dictionary based on the input sequence.

Here is the pseudocode for the LZW encoding algorithm:

```
MAX_TABLE_SIZE=2^(bit_length)   //bit_length is number of encoding bits
initialize TABLE[0 to 255] = code for individual characters
STRING = null
while there are still input symbols:
    SYMBOL = get input symbol
    if STRING + SYMBOL is in TABLE:
        STRING = STRING + SYMBOL
    else:
        output the code for STRING
        If TABLE.size < MAX_TABLE_SIZE:   // if table is not full
            add STRING + SYMBOL to TABLE // STRING + SYMBOL now has a code
        STRING = SYMBOL
output the code for STRING
```

The bit length of the output is a user defined parameter, usually in the range 9 to 12. The table size, defined in the pseudocode as MAX_TABLE_SIZE, depends on the bit length, and is $2^{(\text{bit length})}$.

Note that every time the algorithm adds a new string and code to the dictionary, it also outputs a code. Also, as the dictionary grows, the lengths of the strings it holds increases. Finally, the encoding table, or dictionary, computed during the encoding process does not need to be explicitly transmitted.

Example encoding

The following table illustrates the encoding algorithm for the input sequence "abbbab". It assumes 1-character sequences are already encoded in the table with their ASCII codes. For example, A has a code of 65, B has a code of 66, a has a code of 97, b has a code of 98. (See www.asciitable.com for the extended ASCII codes.)

Note: The first row corresponds to initialization prior to the while, subsequent rows correspond to final values after each iteration of the while loop.

Encoding					
STRING	SYMBOL	STRING + SYMBOL	STRING+SYMBOL in TABLE?	OUTPUT	TABLE update
null					
a	a	a	Y		
b	b	ab	N	97	ab: 256
b	b	bb	N	98	bb: 257
bb	b	bb	Y		
a	a	bba	N	257	bba: 258
ab	b	ab	Y		
				256	

Decoding/Decompressing

Decompression works in the reverse fashion to compression, converting integer codes into the strings they represent. The decompression process for LZW is also straightforward to code, although a little more involved to understand. In addition, it has an advantage over static compression methods because no dictionary or other overhead information is necessary to be transmitted for the decoding algorithm. A dictionary identical to the one created during compression is reconstructed during the decompression process. Both encoding and decoding programs must start with the same initial dictionary, in this case, all 256 extended ASCII characters.

Here is how it works. Mirroring the process carried out by the encoder, every time the decoder extracts a string from the dictionary using a code, it adds a string to the dictionary, consisting of the previous string and the first character of the new string, with an updated code index. So the decoder is adding strings to the dictionary one step behind the encoder.

The LZW decoder first reads an input code (integer) from the encoded sequence, looks up the code in the dictionary by using it as an index, and outputs the string associated with the code. Thereafter, the decoder reads in a new code, finds the new string indexed by this code, and outputs it. The first character of this new string is concatenated to the previously decoded string. This new concatenation is added to the dictionary with an incremented code (simulating how the strings were added during compression). The decoded new string then becomes the previous string, and the process repeats.

When the decoder receives a code that is not already in its dictionary, it can be shown that the new string corresponding to this code consists of the previously decoded string with the first character of the previously decoded string appended. Each time it reads in a code that does not exist in the dictionary, it must add the code and the corresponding string to the dictionary.

Here is the pseudocode for the LZW decoding algorithm:

```
MAX_TABLE_SIZE=2^(bit_length)
initialize TABLE[0 to 255] = code for individual characters
CODE = read next code from encoder
STRING = TABLE[CODE]
output STRING
while there are still codes to receive:
    CODE = read next code from encoder
    if TABLE[CODE] is not defined:      // needed because sometimes the
        NEW_STRING = STRING + STRING[0] // decoder may not yet have code!
    else:
        NEW_STRING = TABLE[CODE]
    output NEW_STRING
    if TABLE.size < MAX_TABLE_SIZE:
         add STRING + NEW_STRING[0] to TABLE
    STRING = NEW_STRING
```

Example decoding

Consider decoding the code sequence generated by the encoding example above:
97 98 257 256

Note: The first row corresponds to initialization prior to the while loop, and each subsequent row corresponds to final values after an iteration of the while loop.

Decoding					
CODE	TABLE[CODE] defined?	STRING	NEW_STRING	OUTPUT	TABLE update
97		a		A	
98	Yes	b	b	B	256: ab
257	No	bb	bb	Bb	257: bb
256	Yes	ab	ab	Ab	258: bba

Tasks for the assignment

You have to implement the LZW algorithm and demonstrate it with fixed bit-length coding. However, the bit length N should be a command line argument. In practice, even though you will be using an N-bit representation for your codes (where N <= 16), you will treat each code as a primitive data type.
For example, in Java you can use a short (16 bits) or an int (32 bits) variable to store each code.

1. Encoding:

- Implement the encoding algorithm. Initially print the encoded data as integers, so it is easy to debug your code. For the "abbbab" input from our example, the encoded data should be 97, 98, 257, 256. You can use this also as the input for the decoder initially.
- The input will be an ASCII text file, whose name is specified as a command line argument, followed by the specified bit length N. For our example, when the input is in the file “input1.txt” and the specified bit length is 12, the command line can be: java Encoder input1.txt 12.
- Each encoded output should be saved as 2 bytes in a file. For the given example, the encoded output of 97 98 257 256 should be saved, in 16-bit format, as 00000000 01100001 00000000 01100010 00000001 00000001 00000001 00000000. Note that each block is a byte. In Java, you can use the UTF-16BE character encoding to write the integer output to file.
- The output filename should have the form <Input File Name without extension> + “.lzw”. For an input file named “input1.txt”, the output file would be named “input1.lzw”.

If you are using Linux, you can use xxd -b <FileName> command to read the byte data in the output file, or you can use online tools like http://www.fileformat.info/tool/hexdump.htm to view the data in hexadecimal format.

2. Decoding:

- Implement the decoding algorithm. To test your decoder initially, get the encoder's output directly as numbers. For our example, the input for the encoder initially will be 97, 98, 257, 256 and the output from the decoder will be “abbbab”.
- Use the encoded file generated by the encoder as the input to the decoder. The command line

arguments should be specified in the order: <Encoded File Name> <Bit Length>. For our example, this would be: java Decoder input1.lzw 12.
- Store the decoded output in a file named <File Name without extension> + "_decoded.txt". So for our example, the output of the decoder would be stored in the file “input1_decoded.txt”.

If you are using Linux, you can use the ‘diff’ command to check whether the input file and output file have identical content.

Example inputs and outputs will be provided on Moodle so you can ensure correctness of your output formats. You are responsible for testing the correctness of your programs.

Grading

Encoding implementation:	35 points
Decoding implementation:	35 points
Storing encoded data in specified format:	15 points
Code design, comments, and README:	15 points

Additional Information

While we have fixed the bit length of the encoding to be N bits, in practice there are several enhancements that can improve the compression performance of the algorithm.

The first is to read and write codes as N-bit integers, where N is determined by the maximum code size. This could lead to codes that are not aligned on byte boundaries. A second enhancement is to read and write codes as variable-length codes, starting with 9-bit codes and growing to longer length codes as the dictionary grows until the maximum bit length is reached. You will not be implementing these enhancements for this assignment, but can implement them for your experimentation and fun!

Submission Guidelines

Please submit your source code along with a README file on Moodle using the submission link. See the syllabus for late policies.

Your submission must include all your source code and a brief report as a README file. Your submission should NOT include any IDE-specific project files, any compiled files, or any executables. Every file should have your name in a comment line at the top. Your README file should have a brief description of your program design and data structure design, the breakdown of the files, a summary of what you think works and fails in your program, which programming language and compiler version you used, and information on how to run your program.

You will submit your project on Moodle. You should submit a single zip or tar file containing your source code files and README file. You can submit your project multiple times; only the most recent project submission will be graded. The most recent project submission will also be used to compute late days, if any.

Final Warning

A project that does not follow the submission guidelines will receive a 10 point deduction. Proper submission is entirely **your responsibility**. Contact the TA if you have any doubts whatsoever about your submission. Do **NOT** submit your project via email. Be sure to keep copies of your files and **do not change them after submitting**. After grades are posted, you have exactly one week to resolve all problems. After that week is up, all grades are final.

IMPORTANT: Please observe the academic integrity guidelines in the syllabus, and submit your own work. Please contact the instructor if you have any questions.