95-771 Data Structures and Algorithms for Information Processing Project 5

Project 5 LZW due Thursday, December 3, 2026 at 11:59:00 PM

Warning: This assignment will take longer than the others that we have done. Please review soon and have questions for class.

Topics: Lempel-Ziv Welch Compression

Write a Java implementation of the Lempel-Ziv Wech compression algorithm. Your program will be able to compress and decompress ASCII and binary files.

Run your solution to compress and decompress shortwords.txt (on the schedule). After testing on shortwords, run compress and decompress on words.html (also on schedule but much larger than shortwords.txt). For another text file, run your solution on CrimeLatLonXY1990.csv (on the course schedule).

For the binary test case, be able to compress and decompress the video on the schedule (01_Overview.mp4). This is a video that I created for another course. You are not responsible for the material in this video. But you do need to be able to compress it and decompress it. Simply download it to your own system and use it as a data file for your LZW solution.

When LZW compression calls for the use of a table, you will use a hash map. This hash map will be implemented with an array of linked lists. You need to write your own hash table and supply your own linked list. The array will provide for 127 linked lists (indexes from 0 to 126). The design of the linked list data field is in your hands.

Note that there are two distinct types of searches going on in LZW. On the one hand, we want to store strings so that we can later look them up and find their correseponding 12-bit values (for LZW compression). On the other hand, we are presented with a 12-bit code word and need to quicky find the corresponding string (for LZW decompression). Your array of linked list is appropriate for putting and getting a value based on a string key. A simple array is appropriate for quickly retrieving a string given a 12-bit key. We will not be using Java's HashMap or TreeMap classes in this program.

In order to execute your program for compression the user will type the following:
java LZWCompression -c shortwords.txt zippedFile.txt

And to decompress the program is run with the following command:
java LZWCompression -d zippedFile.txt unzippedFile.txt

In this example, unzippedFile.txt now has the exact same contents as shortwords.txt.

The user may also decide to show verbose output with the –v switch. The following commands shows the total number of bytes read and the total number of bytes written as the the program compresses and decompresses the file:

java LZWCompression -c –v shortwords.txt zippedFile.txt
bytes read = xxx, bytes written = yyy
java LZWCompression -d –v zippedFile.txt unzippedFile.txt
bytes read = xxx, bytes written = yyy

Your program will use LZW 12-Bit compression. The input file to the compression algorithm will be read in 8-bit bytes. The output file will be written in 12-bit chunks. For example, suppose that the input file contained one byte (8 bits) of data. Your program would read this one byte and write two bytes to the output file. Only the first 12 bits of these 16 bits would be meaningful. For another example, suppose that your input file contained 2 bytes (16 bits) of data. The compressed output file would contain 3 bytes of data. This is because the two 8-bit bytes will compress to two 12-bit chunks. The two 12-bit chunks are contained in 24 bits, or three bytes. It is highly recommened that you test your code with files that are one or two or three bytes in length.

To handle large files, those that overflow the 12-bit table size, your program will detect overflow and generate a brand new table and begin processing anew. You will need to do this in order to handle words.html. (Hint: get things working with small files first.)

Helpful Programs and the LZW Algorithm

Test.java demonstrates some of the issues encountered when moving bytes around with Java. This needs to be studied.

The program CopyBytes.java will be of help when reading and writing files. Note that this program works when reading and writing both binary and ASCII files. This code is a good starting point for this project.

The program GZipDecompress.java shows the way that an application programmer can decompress a file in Java (compression may be handled similarily). Please note that we are not using this approach in this project. We need to implement compression and decompression ourselves. I included it here so that you are aware of it. The program uses a filter design that we are not going to use – unless you are ambitious and want to tackle that issue as well. There will be no extra credit for a filter design.

An example program illustrating how Java represents binary data appears next.

```
// Some notes on Java's internal representation:

// After reading a byte (8 bits) and assigning the byte to a char variable
// (16 bits) we need to clear the uppermost 8 bits in the char since the byte
// will sign extend into the char.

public class Test {

  public static void main(String a[]) {

    byte b = -1;                // b = 0xFF   =   11111111 (8 bits)
    char c = (char)b;           // c = 0xFFFF =   1111111111111111 (16 bits sign extension)
    c = (char)(c & 0xFF);       // c = 0x00FF =   0000000011111111 (remove the extra bits)
    int t = c;                  // t = 0x000000FF = 0000-000011111111 (32 bits)
    System.out.println(t);      // display 255

    b = (byte)255;              // b = 0xFF   =   11111111 (8 bits)
    c = (char)b;                // c = 0xFFFF =   1111111111111111  (16 bits sign extension)
    c = (char)(c & 0xFF);       // c = 0x00FF =   0000000011111111  (remove extra bits)
    t = c;                      // t = 0xFF   =   000-000011111111 (32 bits)
    System.out.println(t);      // displays 255
    System.out.println();

    // In the for loop below, the 16 bits in ch are not converted to an ASCII integer.
    // For example, when ch == 0000000000000000, there is no attempt to convert to ASCII 0
    // The screen may misbehave because some bit sequences are not printable.

    for(char ch = 0; ch <= 255; ch++) {
        System.out.print("" + ch);
    }
    System.out.println();


    // In this for loop, however, the integer is converted to an ASCII representation.
    // For example, when smallInt == 65, two ASCII characters are generated: '6' and '5'.
    // No misbehavior will occur here.

    for(int smallInt = 0; smallInt <= 255; smallInt++) {
        System.out.print("" + smallInt);
    }

    System.out.println();
    // Place a char into a String with no conversion.
    String test = "" + (char)0;
    System.out.println("Test = " + test);

    // Place an int into a String with conversion to ASCII.
    String test2 = "" + 0;
    System.out.println("Test2 = " + test2);

    // So, String objects may hold non-printable sequences of bits.

  }
}
```

The IO classes that are important for project four are exemplified below:

```
// copy a binary or text file
import java.io.*;
public class CopyBytes {
public static void main( String args[]) throws IOException  {
        DataInputStream in =
              new DataInputStream(
                 new BufferedInputStream(
                      new FileInputStream(args[0])));
        DataOutputStream out =
              new DataOutputStream(
                 new BufferedOutputStream(
                      new FileOutputStream(args[1])));
        byte byteIn;
        try {
              while(true) {
                 byteIn = in.readByte();
                 out.writeByte(byteIn);
              }
           }
           catch(EOFException e) {
                 in.close();
                 out.close();
           }
     }
}
```

The compression algorithm that you will implement is as follows:

```
LZW_Compress(){
  enter all symbols in the table;
  read(first character from w into string s);
  while(any input left){
    read(character c);
    if(s + c is in the table)
      s = s + c;
    else {
          output codeword(s);
          Enter s + c into the table;
          s = c;
    } // end if/else
  } // end while
  output codeword(s);
}
```

The decompression algorithm that you will implement appears as follows:

```
LZW_Decompress(){
  enter all symbols into the table;
  read(priorcodeword) and output its corresponding character;

  while(codewords are still left to be input){
    read(codeword);
    if(codeword not in the table)  {
      enter string(priorcodeword) + firstChar(string(priorcodeword)) into the table;
      output string(priorcodeword) + firstChar(string(priorcodeword));
    }
    else {
      enter string(priorcodeword) + firstChar(string(codeword)) into the table;
      output string(codeword);
    }
    priorcodeword = codeword;
  }
}
```

The following classes may not be used in Homework5. You should, however, be familiar with this approach.

```
// Use GZIP decompression to decompress a file          Eckel Chapter 11

import java.io.*;
import java.util.zip.*;

public class GZIPDecompress {

  public static void main(String args[]) throws IOException {

    BufferedReader in = new BufferedReader (
                        new InputStreamReader(
                           new GZIPInputStream(
                              new FileInputStream("test.gz"))));


    String s;
    while((s = in.readLine()) != null) System.out.println(s);

    in.close();
  }
}
```

Sub assignment:

Before writing a solution, I strongly suggest that you write a simpler program that exercises bit manipulation and file I/O. That is, write a program that reads an ASCII or binary file (one byte at a time) and writes to an output file 12-bit chunks that represent the input. If, for example, there is one letter (8 bits) on the input file, the output file would have 16 bits (the leftmost 12 bits of the 16 are active). If the input file holds two letters then the output file will contain exactly 24 bits. This is because each letter expands to 12 bits and two 12 bit chunks fits exactly into three 8 bit bytes. Call this program BadCompressor.java.

Write another program that reads the 12-bit chunks and writes back the original file. Call this program BadDeCompressor.java. If you are unable to write these programs then you probably will not be able to solve the larger problem. If you have trouble with the final solution, turning in this code will receive partial credit (say, 50%). This should be done as a separate activity. It stands on its own as an interesting exercise in bit manipulation.

Suppose your input file contains the single letter "A". The input file will contain 01000001. After the BadCompressor program runs, a new file will contain 0000010000010000. The last four bits are just padding. They will be ignored when read. Run BadDeCompressor.java on that file. The new output file will contain 01000001.

Suppose your input file contains the pair of letters 'AB'. The input file will contain 0100000101000010. After the BadCompressor program runs, a new file will contain 000001000001000001000010. There are no padding bits. This is a 3-byte

file, there are two 12-bit chunks in 3 bytes, exactly. Run BadDeCompressor.java on that file. The new output file will contain 0100000101000010.

I recommend working with an array of three bytes. That is 24 bits. Just enough to hold two 12-bit quantities. You will always read and write 8 bits at a time to a file. The three byte array will act as your internal buffer.

After you have BadCompressor.java and BadDeCompressor.java running on simple ASCII files, try it with binary files. It should expand input files and then return them to their original size. If it does not work for binary files, double check your code to make sure that you are not allowing sign bit extension. If you are not familiar with the distinction between ASII and binary files, here is a nice discussion: https://perlmaven.com/what-is-a-text-file

To view binary files on a Mac, you might use xxd. On a Windows machine, you might try powershell. These programs will provide you with a hex dump of the file. Each hex digit represents 4 bits in the file. I did not use these tools when I solved this program.

Submission Requirements

(1) Submit LZWCompression.java and any other Java files that are required by LZWCompression.java to the assignment section of Canvas.

(2) Within the comments of the main routtine of LZWCompression, describe how your program is working on ASCII files and on binary files. Explain whether your program works for both cases and state the degree of compression obtained on words.html, CrimeLatLonXY1990.csv, and 01_Overview.mp4.

(3) If you were not able to complete LZW compression and decompression, turn in the Sub assignment for partial credit (BadCompressor and BadDecompressor). If you did complete LZW compression and decompression, you need not turn in the sub assignment.

Hint: If ASCII files work and binary files do not, double check any logic that performs automatic sign extension. We want no sign extension here. Java loves to extend the sign bit.

LZW Supplement Prepared by DSA TA Li Zhu

Input: "ABABABA"

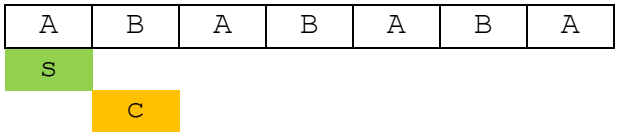
Input file as in binary representation:

0	1	0	0	0	0	0	1	A (65)
0	1	0	0	0	0	1	0	B (66)
0	1	0	0	0	0	0	1	A (65)
0	1	0	0	0	0	1	0	B (66)
0	1	0	0	0	0	0	1	A (65)
0	1	0	0	0	0	1	0	B (66)

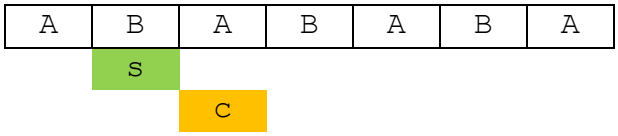
0	1	0	0	0	0	0	1

A (65)

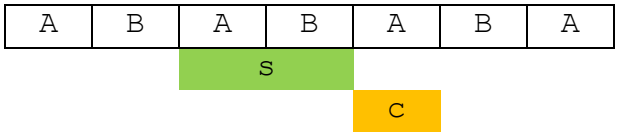
Snapshots when hitting the line output codeword(s)



Output A (65) in 12-bit, enter "AB" with value 256 into the table



Output B (66) in 12-bit, enter "BA" with value 257 into the table



Output AB (256) in 12-bit, enter "ABA" with value 258 into the table



Output ABA (258) in 12-bit after hitting EOF.

What we really **want** to write out to local file:

(4 * 12 bits = 48 bits)

0	0	0	0	0	1	0	0	0	0	0	1
0	0	0	0	0	1	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	1	0

What we **actually can** write out to local file:

(6 * 8 bits = 48 bits)

0	0	0	0	0	1	0	0
0	0	0	1	0	0	0	0
0	1	0	0	0	0	1	0

0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	0