Week 4 Lab - Compression

COSC244

1 Introduction

The idea behind data compression is to make a file as small as possible in order to save disk space or to reduce transmission time over a network. Compression can save both time and money. In lectures, you have seen a number of different ways to perform data compression – Huffman codes, Run Length encoding (bit-level and character-level), and Lempel-Ziv encoding. In this lab, we will work with a variant of the Lempel-Ziv algorithm called LZW compression. This algorithm was published in 1984 by Terry Welch. The compression program and three support classes are provided for your use. You will implement the decompression program from a skeleton program we provide.

1.1 Lempel-Ziv-Welch Compression

The basic idea behind LZW compression is to construct a look-up table which maps character sequences to numbers. Initially the table is created containing only a mapping of an eight bit character set (for example ISO-8859-1) to the numbers 0-255. As input gets processed, the look-up table is expanded by adding to it new combinations of characters as they are encountered. Whenever a combination appears again in the input, only a single number needs to be output in its place, thus saving space.

During decompression, the table is constructed as the compressed file is read just as the table was constructed during the compression process. The table does not have to be sent with the compressed file.

1.2 Lab overview

This lab has 3 steps:

1. Understand the LZW algorithm in detail.
2. Become familiar with the provided support classes.
3. Write a decompression program.

The first two steps above are the preparation for this lab and should be done before coming to the lab.

2 Assessment

This lab is worth 1%. The assessment in this lab consists of tracing the decompression algorithm, writing answers to some questions, and one programming exercise.

Tracing the decompression algorithm and completing the written questions constitute the preparation for this lab. This will be marked by the demonstrators at the start of the lab. The preparation questions should be answered **before** coming to the lab. They should be stored electronically in a plain text file with a .txt suffix in your `~/244/04` directory. We have provided a file `lab04-answers.txt` which you can copy from `/home/cshome/coursework/244/start/04` to put your answers in.

We estimate it will take 2-3 hours to properly prepare for this lab.

Make sure you get your work marked by a demonstrator before leaving the lab.

3 Preparation

3.1 LZW Algorithm

Before we can implement the LZW algorithm (or any algorithm), we must understand it in detail. Wikipedia had a good explanation of the algorithm, along with two worked examples and some pseudocode (Lempel-Ziv-Welch on Wikipedia - Feb 13, 2008). The current version doesn't have the pseudocode anymore, so make sure you use the older version. You can access it from the Resources page of the course webpages. We are more interested in the first, than the second example.

During the tutorial in week 3, you worked through the compression algorithm. Now it is time to work through the decompression algorithm. Here is the decompression algorithm slightly modified from Wikipedia.

```
initialise the dictionary with codes 0-255 and their
    corresponding characters;
read a code;
entry = dictionary entry for the code;
output entry;
while (read a code) {
    prev = entry
    if (code exists in dictionary) {
        entry = dictionary entry for the code;
    } else { // code does not exist in dictionary
        entry = prev + prev[0];
    }
    output entry;
    add (prev + entry[0]) to the dictionary;
}
// NOTE:  x[0] means the first character in the string, x
```

In preparation for the lab, use this decompression algorithm to decompress the following string. The numbers represent the binary data being read from a compressed file.

```
84 79 66 69 79 82 78 79 84 256 258 260 265 259 261 263
```

The dictionary is initialised with the extended ASCII character set in positions 0-255. So the first available slot for insertion in the dictionary is 256. The following table gives the relevant portion of the initialised dictionary.

Code	String
66	B
69	E
78	N
79	O
82	R
84	T

Q1 Fill out the table shown below by tracing the execution of the decompression algorithm on the provided input. There is a copy of this table which you can complete in the `lab04-answers.txt` file. You will find it easiest to open up the file in Emacs because Emacs will handle the table in a special way to keep things tidy.

The demonstrator will check that you have completed this table when checking your lab preparation.

code	prev	entry	prev + entry[0]	output	dictionary
					ASCII code in 0-255
84					
79					
66					
69					
79					
82					
78					
79					
84					
256					
258					
260					
265					
259					
261					
263					

3.2 Provided Support Code

A complete implementation of the LZW algorithm is not a trivial task. We have provided you with 3 support classes. These classes allow you to concentrate on the algorithm itself. You do not need to worry about reading and writing bit stream data or how to construct and access the table – we are giving you classes and methods which do this. Open up the documentation for the provided classes in a web browser using this command:

```
$ firefox /home/cshome/coursework/244/start/04/docs/index.html &
```

3.2.1 BitOutputStream

Let's first look at *BitOutputStream*. This class is used to write the compressed output file as binary data. It is used in the *Compress.java* program which you are provided. You will not need this class to write the decompression program, but it is instructive to look at its methods (and you have to answer the following questions).

Click on *BitOutputStream* and look at the constructor:

```
BitOutputStream(OutputStream out, int bitFieldSize)
```

The first parameter specifies where we want the output to go. It is a *java.io.OutputStream* which *Compress.java* assigns to *System.out*. When you run the program later, it will be redirected to a file.

Q2 Briefly describe the second input parameter.

Q3 What is the purpose of the method, `setBitFieldSize()`?

Q4 What is the purpose of the method, `write()`?

Q5 What is the purpose of the method, `close()`?

3.2.2 BitInputStream

Now look at *BitInputStream*. You will use this class to read a compressed file. Click on *BitInputStream* and look at the constructor:

```
BitInputStream(InputStream in, int bitFieldSize)
```

The first parameter specifies the source of input data. It is a *java.io.InputStream*. The skeleton *Decompress.java* assigns it to *System.in*. When you run your decompression program, you will redirect to get input from a file. The second parameter specifies the number of bits to read. This is set to 9-bits in the skeleton program. More on why this is done later in this document.

Q6 What is the purpose of the method, `setBitFieldSize()`?

Q7 What is the purpose of the method, `read()`?

3.2.3 LookUpTable

Look at *LookUpTable*. This class is used to create and manipulate the table containing strings and their binary codes. Click on *LookUpTable* and look at the constructor:

`LookUpTable()`

Q8 Briefly describe what it does.

Q9 What is the purpose of the method, `add()`?

Q10 What is the purpose of the method, `hasString()`?

Q11 What is the purpose of the method, `getCode()`?

The methods, `add()`, `hasString()` and `getCode()`, are used in the compression algorithm. In addition to `add()` the decompression algorithm uses the two methods, `hasCode()` and `getString()`.

Q12 What is the purpose of the method, `hasCode()`?

Q13 What is the purpose of the method, `getString()`?

3.2.4 A Bit More Explanation

A bit more explanation is needed to fully understand the classes, the compression program, and the decompression skeleton program.

The lookup table is initialised with the codes and their corresponding characters for every 8-bit character. We store the individual characters as a *String* so it is easy to continue on with multiple character sequences. The table initially has entries for codes, 0 - 255.

0 - 255 is the range of codes that can be stored in an 8-bit field. All character sequences added to the initialised dictionary must have codes greater than 255. Thus once we want to read or write a character sequence rather than a single character, we need more than 8 bits. Therefore we would need to begin writing 9-bit codes. The previous 8-bit codes are compatible with the 9-bit codes since the most significant bit of the 8-bit codes will be a zero when expressed with 9-bits. If later we got to code 512, we would have to expand to 10-bit codes. And so on.

In a nutshell, the above scheme writes 8-bit codes until we get to 256, then it begins writing 9-bit codes. Once we get to code 512, we expand to 10-bit codes. We continue adding a bit every time the code doubles to another power of two.

Intuitively one might be tempted to start writing 8-bit codes during compression. In fact, it works fine for compression. Do you see any problem for decompression? Think about it. What do you get when you attempt to read codes above 255 as an 8-bit code? How do you know when, for example, 256 is received?

If we try to decompress a file which was compressed beginning with 8-bit codes, we have a problem when we get to the first 9-bit code. That is, we have only read 8 bits of the code instead of the required 9.

The solution we adopt is to start off writing and reading 9-bit codes for 0 - 255. To make things a bit simpler in the provided classes, compression program and decompression skeleton, we use the code, 256, to indicate it is time to expand the input or output bit stream to an additional bit. The code, 256, is a constant defined in *LookUpTable* called `GROW_CODE`. Because 256 serves a special purpose, the first multiple character sequence goes into slot 257 in the *LookUpTable*. The next goes into slot 258, and so on.

Because we are writing 9-bit codes, compression works well until we need to write code 512. That code requires 10-bits. Before writing 512, *Compress.java* writes code 256 (our special marker code), increases the number of bits being written to 10, and then writes 512. From this point on, all codes are written as 10-bit codes. A similar thing happens when we get to code, 1024, 2048, and so on.

When decompressing a file, we begin by reading 9-bit codes. This allows us to reads codes 0 - 511. When code 256 in encountered, we increment the number of bits read and continue reading.

This is why *Compress.java* and the skeleton *Decompress.java* have the line:

```
int bitFieldSize = 9;
```

We have provided the actual line of code which will increment the bit field size for you within the commented pseudocode in *Decompress.java*, so you don't need to figure it out for yourself. It is

```
in.setBitFieldSize(++bitFieldSize);
```

4 Programming Exercise

Assuming you are in your *244* directory, you can start by copying the provided files into the directory you want to work in using this command:

```
$ cp -r /home/cshome/coursework/244/start/04/src 04
```

This copies all the files you will need for this lab. Change into the *04* directory to begin working on them.

```
$ cd 04
```

Start by compiling the compression program using this command:

```
$ javac Compress.java
```

You should be able to compress *smallfile* like so:

```
$ java Compress < smallfile > smallfile.cmp
```

The task which you need to complete for this lab is to implement the decompression algorithm. We have provided a file called *Decompress.java* which contains skeleton code, as well as pseudocode comments which describe the algorithm. A printout of the file is attached as the next to last page of this document.

Your task is to write the code which implements the pseudocode comments. You can test your program by decompressing *smallfile.cmp* using this command:

```
$ java Decompress < smallfile.cmp > smallfile.dcp
```

You can either visually compare *smallfile* to *smallfile.dcp* or type the following command which should show no differences:

```
$ diff smallfile smallfile.dcp
```

Once you have completed this program ask a demonstrator to check your work.

Make sure you get your work signed off by a demonstrator before leaving the lab.

5 Optional Extension Exercise

This part is for those who want a bit more of a challenge. It is not required or marked.

Add some code to *Compress.java* and *Decompress.java* to measure how long the compression or decompression takes, and print this information to *System.err*. You might find the method `System.currentTimeMillis()` helpful here.

After you have that working, add some code to *Compress.java* to make it read from a file instead of *System.in* and write to a file instead of *System.out*. Now add some code to *Compress.java* to calculate the size of the compressed file as a percentage of the original file.

Listing of Decompress.java

```
/**
 * This class performs LZW decompression on data read from stdin.  It
 * reads the compressed input from stdin using a BitInputStream which
 * allows a variable width bit field to be used.
 */
public class Decompress {

   /**
    *  The entry point of the program.  Performs LZW decompression on input
    *  read from stdin, and outputs it to stdout.
    *
    * @param  args command line arguments are not used.
    * @exception  Exception  Throws all exceptions (no error handling done)
    */
   public static void main(String[] args) throws Exception {
      int bitFieldSize = 9; // 8 bits for character set + 1 bit for growcode
      BitInputStream in = new BitInputStream(System.in, bitFieldSize);
      LookUpTable table = new LookUpTable(); // initialise the dictionary
      int code;     // the code read from stdin
      String entry = null; // current entry from dictionary
      String prev = null;  // previous entry from dictionary

      /** process the first code value in the compressed file/stream */
      if ((code = in.read())!= -1) {  // if there is input
         // let entry = table entry for code
         // output entry
      }

      /** process the remaining codes in the compressed file/stream */
      // while there is input
      //    if code is the grow code
      //       in.setBitFieldSize(++bitFieldSize);
      //    otherwise
      //       let prev = entry
      //       let entry = table entry for code
      //       if there wasn't a table entry for code then
      //          let entry = prev + first char of prev
      //       output entry
      //       add prev + first char of entry to the table
   }
}
```