Understanding the Lempel-Ziv Data Compression Algorithm in Java

Learn how to write a Java program that illustrates lossless data compression according to the Lempel-Ziv Compression Algorithm commonly known as LZ77. Also learn about the characteristics of the algorithm that result in data compression.

Published: February 21, 2006
By Richard G. Baldwin

Java Programming Notes # 2440

Preface

This is the first lesson in a new series of lessons that will teach you about data and image compression. The series begins with the Lempel-Ziv Lossless Data Compression Algorithm, commonly known as LZ77.

Future lessons will cover a variety of compression schemes, including:

- Huffman data encoding
- Run-length data encoding
- GIF image compression
- JPEG image compression

The programs that I will provide in these lessons are provided for educational purposes only. If you use these programs for any purpose, you are using them at your own risk. I accept no responsibility for any damages that you may incur as a result of the use of these programs.

Viewing tip

You may find it useful to open another copy of this lesson in a separate browser window. That will make it easier for you to scroll back and forth among the different listings and figures while you are reading about them.

Supplementary material

I recommend that you also study the other lessons in my extensive collection of online Java tutorials. You will find those lessons published at Gamelan.com. However, as of the date of this writing, Gamelan doesn't maintain a consolidated index of my Java tutorial lessons, and sometimes they are difficult to locate there. You will find a consolidated index at www.DickBaldwin.com.

General Background Information

Most of us use data or image compression on a daily basis without even thinking about it. If you use one of the popular zip programs to archive your data, you are using a program that typically implements several different data compression algorithms in combination. If you take pictures with a digital camera, you are creating files that describe images using the JPEG image compression format. You may be able to adjust the parameters on your camera to provide a tradeoff between image quality on one hand and image compression on the other.

The Lempel-Ziv algorithm

LZ77 is the name commonly given to a lossless data compression algorithm published in papers by Abraham Lempel and Jacob Ziv in 1977. This algorithm forms the basis for many LZ variations including LZW, LZSS and others.

A lossless compression algorithm

LZ77 is a *lossless* compression algorithm. What this means is that if you compress a document using the algorithm, and then decompress the compressed version, the result will be an exact copy of the original document. Not all compression algorithms are lossless. The JPEG image compression algorithm, for example, does not produce an exact copy of an image that has been compressed using the algorithm.

A dictionary encoding algorithm

LZ77 is known as a dictionary encoding algorithm, as opposed for example to the Huffman encoding algorithm, which is a statistical encoding algorithm. Compression in the LZ77 algorithm is based on the notion that strings of characters *(words, phrases, etc.)* occur repeatedly in the message being compressed. Compression with the Huffman encoding algorithm is based on the probability of occurrence of individual characters in the message.

Combinations of compression algorithms

Different variations of the LZ algorithms, the Huffman algorithm, and other compression algorithms are often combined in data and image compression programs. For example, numerous sources on the web indicate that commercially available zip programs often incorporate something called **DEFLATE**. According to Wikipedia,

*"**DEFLATE** is a lossless data compression algorithm that uses a combination of the LZ77 algorithm and Huffman coding. It was originally defined by Phil Katz for version 2 of his PKZIP archiving tool, and was later specified in RFC 1951."*

Search for Lempel and Ziv on the Internet

Beyond this brief background description, if you go to Google and search for the keywords **Lempel** and **Ziv**, you will find more online material than you can possibly have the time to read. A particularly useful resource is the animated program named LZ that you can download from Sami Khuri's Visualization and Animation Packages.

Preview

As mentioned above, LZ77 is the name commonly given to a lossless data compression algorithm published in papers by Abraham Lempel and Jacob Ziv in 1977.

Explaining how LZ77 works

The Java program named **LZ77v01** that I will present and explain in this lesson implements a *very close approximation* to theLZ77 compression algorithm. While this program may not match that algorithm in every respect, the match is close enough to satisfy the primary purpose of the program, which is to explain how the LZ77 algorithm works.

Not a production compression program

The program was specifically designed to serve its primary purpose of education. No thought or effort was given to speed, efficiency, memory utilization, or any other factor that would be important in a program written for production data compression purposes. In some cases, the program was purposely made less efficient *(in the name of clarity)* by using two or more statements to accomplish a task that could be accomplished by a single more complex statement.

Sample messages

Two sample messages are hard-coded into the program. You can switch between those messages by enabling and disabling one or the other using comments and then recompiling the program. You can also insert your own message and recompile the program to use your message if you choose to do so.

A collection of Tuple objects

The program first encodes a message into a collection of **Tuple** objects stored in an **ArrayList** object using a very close approximation to the LZ77 lossless data compression algorithm. Then the program decodes the message by extracting the information from the **Tuple** objects and using that information to reconstruct the original message.

*(**Tuple** is the name of a class that I used to encapsulate a three-element tuple containing three values that occur repeatedly in the LZ77 algorithm. I will use the uppercase version, Tuple, to refer to the class or an object of the class, and use the lowercase version, tuple, to refer to the three elements in a more general sense.)*

No conversion to a bit stream

The program does not actually convert the encoded *(compressed)* message into a stream of bits, although it wouldn't be difficult to do so. Since the purpose of the effort was to illustrate the behavior of the LZ77algorithm, and not to write a production data compression program, converting the compressed data into a bit stream was considered to be superfluous. However, the program does provide a theoretical analysis of the amount of compression that would be achieved if the results were converted into a bit stream.

Program output

Quite a lot of output material is displayed on the screen while this program is running. This material is designed to be used in this tutorial lesson to explain the behavior of the LZ data compression algorithm.

Program testing

This program was tested against the demonstration Java program named LZ that can be downloaded from Sami Khuri's Visualization and Animation Packages. Although the source code for that program was not consulted during the development of my program, the animated interactive nature of Khuri's program was extremely helpful to me in understanding the LZ77 algorithm.

The output from my program matches the output from Khuri's program for 60 characters of input data and values of 10 for **searchWindowLen** and **lookAheadWindowLen**. *(Khuri's program is limited to an input data length of 60 characters.)*

The program was tested using J2SE 5.0 and WinXP. J2SE 5.0 or later is required due to the use of generics in the program.

Discussion and Sample Code

The LZ77v01 program

The program named LZ77v01 is presented in its entirety in Listing 24 near the end of the lesson.

I will discuss the program in fragments. The class definition begins in Listing 1.

```
class LZ77v01{
  //Use comments to enable and disable and to
```

```
select
  //between the following demonstration
messages.
/*
  String rawData = "BAGHDAD, Iraq Violence
increased "
  + "across Iraq after a lull following the
Dec. 15 "
  + "parliamentary elections, with at least two
dozen "
  + "people including a U.S. soldier killed
Monday in "
  + "shootings and bombings mostly targeting
the Shiite-"
  + "dominated security services. The Defense
Ministry "
  + "director of operations, Brig. Gen. Abdul
Aziz "
  + "Mohammed-Jassim, blamed increased violence
in the "
  + "past two days on insurgents trying to
deepen the "
  + "political turmoil following the elections.
The "
  + "violence came as three Iraqi opposition
groups "
  + "threatened another wave of protests and
civil "
  + "disobedience if allegations of fraud are
not "
  + "properly investigated.";
*/
  String rawData = "Miss Kalissippi from
Mississippi is a "
  + "cowgirl who yells yippi when she rides her
horse in "
  + "the horse show in Mississippi.";
```

Listing 1

The class begins by hard-coding two sample messages into the program. The first message is a news message copied directly from an Internet web site. The second message is a message from my own imagination that was designed specifically to illustrate important concepts of the LZ77 algorithm.

You can select between these two messages by disabling one or the other through the use of comments and then recompiling the program. Alternately, you can insert your own message and test the behavior of the algorithm with a message of your choosing.

Miscellaneous instance variables

Several instance variables are declared in Listing 2.

```
  //A collection of Tuple objects
  ArrayList <Tuple>encodedData = new
ArrayList<Tuple>();

  //A reference to a single Tuple object
  Tuple thisTuple;

  //A substring that represents the search
window.
  String searchSubstring;

  //Working variables
  int matchLen;
  int matchLoc;
  int charCnt;
  int searchWindowStart;
  int lookAheadWindowEnd;
```

Listing 2

The class named **Tuple** is a simple wrapper class for which you can find the source code in Listing 24. The purpose of an object of the **Tuple** class is to encapsulate three values that are critical to the behavior of the LZ77 algorithm. I will have more to say about this later.

The search window and the lookAhead window

The LZ77 algorithm is based on the use of two sliding contiguous windows on the data being compressed.

```
  //Modify the following values to change the
length of
  // the search window and/or the lookAhead
window.
  // Optimum values are one less than even
powers of 2.
  int searchWindowLen = 31;
  int lookAheadWindowLen = 7;
```

Listing 3

You can modify the size of those windows by modifying the initialization values of the two variables shown in Listing 3. I will have more to say about setting the respective sizes of the two windows shortly.

Compression on-the-fly

The LZ77 algorithm compresses the message *on-the-fly* as a pointer to the current location moves through the message from beginning to end.

The lookAhead window

The *lookAhead window* consists of a specified number of characters ahead of the current location. The lookAhead window has an initial size as specified in Listing 3 and decreases to zero as the *current-location pointer* approaches the end of the message.

The search window

The search window consists of a specified number of characters, *(which have already been traversed)*, behind the current location. The search window has an initial size of zero characters and increases to the specified value shown in Listing 3 as the current-location pointer moves through the data.

The output is a collection of Tuple objects

As each new character is examined, a **Tuple** object is created in the output containing three values. Thus, the output from the compression process consists of a collection of **Tuple** objects.

> *(Alternatively, each output tuple could consist of a group of bits of a specified size that encapsulate the three required values.)*

If the message is compressible *(meaning that the message consists of a sequence of repetitive matching strings)*, the number of tuples in the output will be less than the number of characters in the message. This is accomplished by causing the contents of each tuple to point to the previous occurrence of a matching string of characters rather than to simply repeat the string of characters. Upon encountering such a tuple, the decompression process retrieves the matching string of characters from the previously-reconstructed output characters, and appends the matching string onto the end of the output character stream.

The compression factor

An output tuple always contains one character plus two other values. Thus, the size of a tuple *(consisting of a group of bits of a specified size)* will always be larger than the number of bits required for a single character in the input message.

> *(The hope is that each output tuple will contain sufficient information, in addition to the character that it contains, to cause several characters to be created in the decompressed output. This will only be true if the input message contains repetitive strings, such as repetitive words or phrases. For example, messages composed in the English language often contain many occurrences of words such as* ***the****,* ***and****,* ***in****,* ***but****,* ***for****,* ***to****,* ***be****,* ***than****, etc.)*

Compression greater than 1.0

In order for the compression factor to be greater than 1.0, the number of tuples in the output must be less than the number of characters in the input. Further, that difference must be sufficiently large to compensate for the fact that the size of a tuple is greater than the size of a single character.

(I will show you examples later where the compression factor is greater than 1.0 a well as an example where the compression factor is less than 1.0.)

The LZ77 compression algorithm

The implementation of the compression algorithm is rather straightforward and can be described in the following steps.

1. Set the *current-location pointer* to the beginning of the message.

2. Find the longest match that occurs in the *search window* for the beginning sequence of characters in the *lookAhead window*.

3. Process the possible match:

   a. If there is no match for even the first character in the lookAhead window, output a tuple containing the following three values and increment the current-location pointer by 1:

   - **offset:** 0
   - **stringLen:** 0
   - **nextChar:** The unmatched first character in the lookAhead window.

   b. If there is a match, output a tuple containing the following three values and increment the current-location pointer by the length of the match plus 1:

   - **offset:** The location of the beginning of the match in the *search window* relative to the current location. *(Note that even though this pointer is pointing backwards towards the beginning of the message, it is recorded as a positive value.)*
   - **stringLen:** The length of the match.
   - **nextChar:** The first non-matching character in the *lookAhead window* following the characters that match.

4. Go back to step 2 and repeat the process until the end of the message is encountered, saving the collection of output tuples in some appropriate container.

The compressed message

When the end of the message is reached, the collection of tuples contains the compressed message. The information in the **Tuple** objects can be converted into a stream of bits, written into a file, or otherwise disposed of in whatever way is appropriate.

*(In a production compression program, the tuples could be written into a stream of output bits on-the-fly without the use of an intermediate collection of **Tuple** objects.)*

Decompressing the compressed message

Decompression of the compressed message is even easier than compressing the message. I will describe the decompression process in more detail later.

The required number of bits for a tuple

The number of bits required for the first two values in the tuple depends on the respective sizes of the *search window* and the *lookAhead window*.

> *(I am assuming that eight bits are required for the third value in the tuple, which is an eight-bit character.)*

For example, the number of bits required for the first value in the tuple depends on the maximum possible value of the pointer that points to the matching string in the search window relative to the current location. This, in turn, depends on the size of the search window.

Similarly, the number of bits required for the second value in the tuple depends on the maximum value of the length of the matching string. This, in turn, depends on the size of the lookAhead widow.

Optimizing the window sizes for bit size considerations

From the viewpoint of the number of bits required for the first two values in the tuple, the optimum values for these two window sizes are one less than an even power of two. For example, a window size of 7 can be stored in three bits whereas a window size of 8 requires four bits. If you are going to set the size to 8, you might as well set it to 15.

Optimizing the *lookAhead window* size for compression

From a compression viewpoint, the optimum sizes for the two sliding windows depend on the nature of the data being compressed. In general, I would think that the lookAhead window needn't be very large for a message written in the English language. Except in rare cases, the probable length of a matching string in such a message is unlikely to be greater than about eleven characters *(Mississippi for example)*. In that case, a lookAhead window size of fifteen characters *(four bits)* might be appropriate.

Optimizing the *search window* size for compression

The optimum size of the search window is more difficult to predict. On one hand, for long messages, the larger the search window, the greater is the probability that a match will be found in many cases. On the other hand, the larger the search window, the more bits will be required for the first value in the tuple regardless of whether or not a match is found.

This is clearly a situation where a tradeoff is in order. Increasing the probability of finding a match improves the compression, but increasing the required size for all tuples decreases the compression. A balance must be found between the two.

Some experimental results

At this point, I'm going to show you some experimental results to help you understand the explanation of the compression algorithm given above. These experimental results were produced by using the program named **LZ77v01** to compress the message shown in Figure 1.

```
Miss Kalissippi from Mississippi is a cowgirl
who yells yip
pi when she rides her horse in the horse show
in Mississipp
i.
Figure 1
```

The message in Figure 1 was displayed with 59 characters per line.

The experimental output

Figure 2 shows three consecutive lines of screen output near the middle of the run.

```
ssippi is a cowgirl who yells y - ippi wh -
29,5,w - ippi w
 is a cowgirl who yells yippi w - hen she -
16,1,e - he
s a cowgirl who yells yippi whe - n she r -
0,0,n  - n
Figure 2
```

The search window contents

The red text in the first line in Figure 2 shows the contents of the 31-character search window at that point in the program. The position of the search window in the message corresponds to the red text shown in Figure 1.

The lookAhead window contents

The blue text in the first line in Figure 2 shows the contents of the 7-character lookAhead window at that point in the program. The position of the lookAhead window in the message corresponds to the blue text shown in Figure 1.

> *(The dash and the spaces on either side of the dash between the red text and the blue text in Figure 2 were placed there to provide a visual separation between the contents of the search window on the left and the contents of the lookAhead window on the right. They are not part of the message.)*

The position of the *current-location pointer*

The *current-location pointer* is at the juncture of the red and blue characters in Figure 1 at that point in the program.

Tracing the algorithm

For this iteration of the program, the algorithm begins with the first blue **"i"** in the lookAhead window in Figure 2 and determines how many consecutive characters in the lookAhead window can be matched up with a string of characters in the red search window of Figure 2.

Where is the match?

The result is that beginning with the third red character in Figure 2, there are five consecutive characters, *(ippi),* that match the first five characters in the blue lookAhead window *(including the space character).*

The first of those five matching characters occurs 29 characters to the left of the current location.

Finally, the first character in the lookAhead window that doesn't match the string in the search window is the **"w"** character.

Create a Tuple object

The program creates a **Tuple** object that encapsulates the following three values and saves the **Tuple** object in the collection of **Tuple** objects:

- **offset:** 29 *(distance back to the matching string relative to the current location)*
- **stringLen:** 5 *(length of the matching string)*
- **nextChar:** w *(first non-matching character in the lookAhead window)*

The material in Figure 2 is separated into four columns with the columns separated by dashes. The three values encapsulated in the **tuple** are shown in black in the first line, third column of Figure 2.

Reconstructing from the tuple information

The fourth column in Figure 2 shows the result of using the information contained in the tuple to reconstruct a small portion of the original message. These are the characters that will be appended to the output when the tuple is used later to decompress the entire compressed message.

A very fruitful tuple

This particular tuple was a very fruitful tuple from a compression viewpoint. The expansion of the contents of this tuple would cause six new characters, *(ippi w)*, to be appended to the output when the message is decompressed.

Move the current-location pointer

Because those six characters of input data have now been handled, the *current-location* pointer is moved six characters to the right in the execution of the compression algorithm. The new position for the *current-location pointer* is between the **"w"** and the **"h"** characters near the left side of the second line in Figure 1.

> *(You can also see that by looking at the last character in the search window and the first character of the lookAhead window for the second line in Figure 2. Once again, the dashes in Figure 2 are there solely to separate the columns. They are not part of the message data.)*

Not quite as fruitful

The processing of the data in the lookAhead window for the second line in Figure 2 wasn't nearly as fruitful as for the first line. Only the **"h"** character matched a character in the search window as indicated by the middle value of **"1"** in the tuple contents in the fourth column in Figure 2. In this case, the tuple contents were:

- **offset:** 16
- **stringLen:** 1
- **nextChar:** e

The fourth column in the second line of Figure 2 shows that upon decompression, this tuple would cause the string **"he"** to be appended to the output.

A break-even proposition

For *search window* and *lookAhead window* sizes of 31 characters and 7 characters respectively, sixteen bits would be required to contain the data stored in this tuple *(assuming 8-bit character data)*. From a compression viewpoint, a 16-bit tuple that produces two characters in the output is a break-even proposition.

A net loss of 8 bits

The third line in Figure 2 shows the worst of all worlds. The first character, **"n"**, in the lookAhead window doesn't match even one character in the search window. Thus, a tuple was constructed that encapsulates the following values:

- **offset:** 0
- **stringLen:** 0
- **nextChar:** n

Only one character is produced

The fourth column in the third line of Figure 2 shows that upon decompression, this tuple would cause only the single character **"n"** to be appended to the output. In terms of compression, a 16-bit tuple that transfers only one 8-bit character to the decompressed output represents a net loss of eight bits.

Now let's get back to a discussion of the program code.

The main method

Listing 4 shows the **main** method in its entirety.

```
  public static void main(String[] args){
    new LZ77v01().doIt();
  }//end main

Listing 4
```

The **main** method simply instantiates a new object of the class and invokes the **doIt** method on the object.

The doIt method

The **doIt** method begins in Listing 5.

```
  void doIt(){

    //Display a scale on the screen.

System.out.println("12345678901234567890123456789"
                   +
"12345678901234567890123456789");

    //Display the raw data, 59 characters to the
line.
    display59(rawData);
    System.out.println();//blank line

Listing 5
```

Listing 5 displays a numeric scale across the top of the screen *(to assist me in my publication efforts)* and then invokes the method named **display59** to cause the message being compressed to be displayed 59 characters to the line *(as shown in Figure 1)*.

The display59 method

The method named **display59** can be viewed in Listing 24 near the end of the lesson. This is a very simple utility method that shouldn't require an explanation.

Process the message

Listing 6 shows the beginning of a **while** loop that is used to step through and compress the message one character *(or hopefully one substring)* at a time.

```
    charCnt = 0;
    while(charCnt < rawData.length()){

Listing 6
```

> *(If it is necessary to compress the message one character at a time, the compressed output will be larger then the uncompressed input. In order for compression to be fruitful, it must be possible to frequently include substrings in the output tuples that constitute the compressed message so that the number of output tuples will be less than the number of input characters.)*

The beginning of the search window

Listing 7 establishes the beginning of the search window during each iteration of the **while** loop.

```
      searchWindowStart = (charCnt-
searchWindowLen >=0)
                                ? charCnt-
searchWindowLen
                                : 0;

Listing 7
```

> *(In case you don't recognize this syntax, this statement is sometimes called a **conditional** statement. I often call it a **cheap if** statement.)*

If the expression in the parentheses evaluates to true, the entire expression on the right of the assignment operator returns the value produced by the expression following the question mark. Otherwise, the expression on the right of the assignment operator returns the value produced by the expression following the colon.

A sliding search window

The purpose of using a conditional statement in this case is to cause the beginning of the search window to be the first character in the message until the search window reaches the specified size given by the value of **searchWindowLen**. Following that, the beginning of the search window follows the *current-location pointer* at a fixed distance from the *current-location pointer* causing the size of the search window to remain constant.

A sliding lookAhead window

Listing 8 does the same thing for the lookAhead window, except in reverse.

```
      lookAheadWindowEnd =
           (charCnt+lookAheadWindowLen <
rawData.length())
                              ?
charCnt+lookAheadWindowLen

:rawData.length();
```

Listing 8

The lookAhead window starts out at full size *(assuming the length of the message exceeds the size of the lookAhead window)* and then shrinks to zero at the end of the message.

Display search window and lookAhead window contents

Listing 9 displays the material shown in red and blue in the first line of Figure 2.

```
      //Display the contents of the search
window.
      System.out.print(rawData.substring(

searchWindowStart,charCnt) + " - ");

      //Display the contents of the
lookAheadWindow on the
      // same line as the search window,
separated by a
      // dash.
      System.out.print(rawData.substring(

charCnt,lookAheadWindowEnd) + " - ");
```

Listing 9

One line of material like the first line in Figure 2 is displayed during each iteration of the **while** loop that began in Listing 6.

> *(Obviously this display is for educational purposes only and has nothing to do with the actual compression of the message.)*

Get a search window substring

Listing 10 extracts a substring from the message that contains the same data as the search window. The purpose is to make it easier to write the code that searches for a match to the characters in the lookAhead window.

```
      if(charCnt == 0){
        //Begin with an empty search window.
        searchSubstring = "";
      }else{
```

```
        searchSubstring = rawData.substring(

searchWindowStart,charCnt);
      }//end else

Listing 10
```

This is one of those cases where I purposely made the program less efficient in order to make the code less cryptic. I could have performed the search on the search window portion of the message itself, but the indexing would have been more cryptic.

Search for a match in the search window

Listing 11 searches to determine the longest substring from the lookAhead window, *(beginning with the first character in the lookAhead window)*, for which there is a matching substring in the search window.

```
      matchLen = 1;
      String searchTarget = rawData.substring(
                                charCnt,charCnt
+ matchLen);
      if(searchSubstring.indexOf(searchTarget)
!= -1){
        //A match was found for the one-
character string.
        // See if the match extends beyond one
character.
        //Limit the length of the possible
match to
        // lookAheadWindowLen
        matchLen++;//Increment length of
searchTarget.
        while(matchLen <= lookAheadWindowLen){
          //Keep testing and extending the
length of the
          // string being tested for a match
until the
          // test fails.
          searchTarget = rawData.substring(

charCnt,charCnt+matchLen);
          matchLoc =
searchSubstring.indexOf(searchTarget);
          //Be careful to avoid searching
beyond the end
          // of the raw data.
          if((matchLoc != -1) && ((charCnt +
matchLen)
                                      <
rawData.length())){
            matchLen++;
          }else{
            //The matching test failed.  Break
```

```
out of the
            // loop.
            break;
          }//end else
        }//end while

Listing 11
```

The **while** loop in Listing 11 continues looping until the matching test fails. When the matching test fails, control breaks out of the loop.

Go back one step

When control breaks out of the **while** loop in Listing 11, the value for **matchLen** has been incremented one time too many, causing the matching test to fail.

```
        //Reduce matchLen to the longest length
that
        // matched.
        matchLen--;

        //We went one step too far increasing
matchLen. Go
        // back and get the location in the
search window
        // where the last match occurred.
        matchLoc = searchSubstring.indexOf(

rawData.substring(charCnt,charCnt+matchLen));

Listing 12
```

Listing 12 begins by reducing the value of **matchLen** by one character to compensate for the overshoot described above.

Get the matching location

Then Listing 12 uses that corrected value of **matchLen** to perform one more search, *(which is guaranteed to succeed)*, to determine the location in the search window that matches the beginning of the substring from the lookAhead window.

Skip matchLen characters in the input message

Listing 13 increases the value of the conditional control variable *(otherwise referred to herein as the current-location pointer)* for the outer **while** loop to cause the process to skip over all of the characters included in the matching substring from the lookAhead window.

```
        charCnt += matchLen;
```

```
Listing 13
```

*(Hopefully, such increases will occur often and cause the number of iterations performed by the outer **while** loop to be much less than the number of characters in the input message. One **Tuple** object is produced for every iteration of the outer **while** loop. For compression to be fruitful, the number of **Tuple** objects produced must be much less than the number of characters in the message being compressed.)*

Instantiate a Tuple object

As a result of finding a match between the substring in the lookAhead window and the characters in the search window, Listing 14 instantiates an object of the **Tuple** class that encapsulates the following information:

- **offset:** Distance from the current location back to the location of the beginning of the match in the search window.
- **stringLen:** Length of the match.
- **nextChar:** First non-matching character in the lookAhead window following the matching substring

```
        //Calculate the offset
        int offset =
                    (charCnt < (searchWindowLen
+ matchLen))
                               ? charCnt -
matchLoc - matchLen
                               :searchWindowLen -
matchLoc;
        //Get and save the next non-matching
character in
        // the lookAhead window.
        String nextChar =

rawData.substring(charCnt,charCnt+1);
        //Instantiate and populate the Tuple
object.
        thisTuple = new
Tuple(offset,matchLen,nextChar);

Listing 14
```

This is the fruitful form of tuple

This is the form of the **Tuple** object that is instantiated for situations like the ones illustrated by the first and second lines in Figure 2.

The offset calculation in Listing 14 is a little cryptic, but shouldn't be beyond the capabilities of those persons who regularly study my Java programming lessons.

The Tuple class

You can view the **Tuple** class in its entirety in Listing 24. The class is a simple wrapper class designed to encapsulate the three values listed above and shouldn't require further explanation.

Save the Tuple object

Listing 13 saves the **Tuple** object in a collection of type **ArrayList**.

```
        encodedData.add(thisTuple);

Listing 15
```

No match was found

Listing 16 shows the **else** clause that goes with the **if** statement at the top of Listing 11.

```
      }else{
        //A match was not found for the next
character.
        //Encapsulate the following information
in a Tuple
        // object.
        //1. 0
        //2. 0
        //3. The non-matching character.
        String nextChar = rawData.substring(

charCnt,charCnt+1);
        thisTuple = new Tuple(0,0,nextChar);
        encodedData.add(thisTuple);
      }//end else

Listing 16
```

The code in the **else** clause in Listing 16 is executed when the first character in the lookAhead window doesn't match any of the characters in the search window.

An unfruitful tuple

Listing 16 instantiates a **Tuple** object containing the information shown in the comments and saves the **Tuple** object in the **ArrayList** collection. This is the form of the **Tuple** object that is instantiated for situations like the one illustrated in the third line in Figure 2. This form of tuple represents a net loss in terms of compression.

Increment the loop counter

Listing 17 increments the counter *(the current-location pointer)* that controls the outer **while** loop that began in Listing 6.

```
      charCnt++;

Listing 17
```

Display the Tuple contents

Listing 18 displays the contents of the **Tuple** object on the same line as the search window and the lookAhead window as shown in Figure 2. The contents of the **Tuple** object are shown in the third column in Figure 2.

```
      System.out.print(thisTuple + " - ");

Listing 18
```

Display text represented by the Tuple object

Listing 19 displays the text represented by the contents of the **Tuple** as shown in Figure 2. This is the material shown in the fourth column in Figure 2.

```
      if(thisTuple.stringLen > 0){
        //The Tuple contains a pointer to a
character or
        // string in the search window.  Expand
and
        // display it.
        int start = charCnt - 1 -
thisTuple.stringLen
                                          -
thisTuple.offset;
        int end = charCnt - 1 -
thisTuple.offset;
        System.out.println(rawData.substring(
                           start,end) +
thisTuple.nextChar);
      }else{
        //The tuple contains a character for
which no
        // match was found.  Display it.
        System.out.println(thisTuple.nextChar);
      }//end else

    }//end while loop

Listing 19
```

The comments in Listing 19 should speak for themselves.

Listing 19 also signals the end of the **while** loop that began in Listing 6.

Message has been compressed

When the **while** that extends from Listing 6 through Listing 19 terminates, the message has been compressed.

The original message has been encoded into a collection of tuples where each tuple contains the following information:

- **offset:** Points backwards in the original message to the beginning of a match relative to the current location. Has a value of 0 if there is no match for the next character *(the first character in the lookAhead window)*.
- **stringLen:** Specifies the length of the match or 0 if there is no match for the next character.
- **nextChar:** The first non-matching character following a match, or the next character if there is no match.

A compressed stream of bits

It would not be particularly difficult at this point to iterate on the collection of **Tuple** objects and to produce a binary bit stream containing these three values.

For example, depending on the values specified for **searchWindowLen** and **lookAheadWindowLen**, the first two values listed above could be encoded into the number of bits required to contain the largest possible **offset** value and the largest possible **stringLen** value. The character from the **Tuple** object could be encoded into eight bits. Then, the set of bits required to describe the tuple could be inserted into a stream of bits.

However, this demonstration program does not produce such a compressed binary bit stream.

Decompress and display the message

Listing 20 shows the beginning of the code used to decompress and to display the compressed message.

```
    StringBuffer reconData = new
StringBuffer();

Listing 20
```

The decompressed message is reconstructed into the **StringBuffer** object that is declared in Listing 20.

The form of the compressed message

The compressed message is represented by a set of tuples where each tuple is a simple structure containing the three values listed above.

Each tuple could be stored in an object, as is the case in this program.

Alternately, each tuple could be stored in a set of contiguous bits in a stream of bits as might be read from a disk file. As near as I have been able to determine, the LZ77 algorithm is not intended to be used with variable length data. Thus, each set of bits that encapsulates a tuple would need to be the same length.

In this program, the tuples are stored in objects instantiated from the class named **Tuple**, so the following explanation of the decompression process will be based on the use of such objects.

The decompression process

The simplicity of the decompression process is one of the strengths of the LZ77 algorithm. The process iterates through the set of **Tuple** objects, sequentially processing one **Tuple** object at a time. For each **Tuple** object, there are two possibilities.

First possibility, no match was found

This is the simpler of the two possibilities. *(However, it doesn't help the compression factor.)* As mentioned earlier, this possibility probably represents a net loss insofar as compression is concerned.

If the value of **stringLen** encapsulated in the **Tuple** object is 0, simply extract the character referred to by **nextChar** and append it onto the end of the message that is being reconstructed.

Second possibility, a match was found

If the value of **stringLen** encapsulated in the **Tuple** object is not 0, this indicates that the **Tuple** object can be used to produce a substring plus the character referred to by **nextChar**. *(Large values for **stringLen** have a positive impact insofar as compression is concerned.)*

Where is the substring stored?

The substring already exists in the message that is being constructed. *(The required substring was constructed earlier in the decompression process.)* All that is necessary at this point is to get a copy of that substring and to append that copy onto the end of the message that is being constructed.

The beginning of the existing substring *(relative to the current location)* is pointed to by the value of **offset**. Just count characters in the direction of the beginning of the message. When you have counted a number of characters equal to the value of **offset**, you will have reached the beginning of the required substring.

The length of the substring is specified by the value of **stringLen**.

Get a copy of the substring

Use the values of **offset** and **stringLen** to extract a copy of the substring from the portion of the message that has already been constructed and append the copy onto the end of the message being constructed.

The single character

Then get the single character from the **Tuple** object and append it onto the end of the message being constructed.

Repeat the process

Repeat the processing of individual tuples until you run out of **Tuple** objects. When you have processed the last **Tuple** object, you will have reconstructed a copy of the original message from the compressed representation of the message.

The decompression code

The code to accomplish the decompression process is shown in Listing 21.

```
    //Get an iterator on the collection of
Tuple objects.
    Iterator <Tuple>iterator =
encodedData.iterator();

    //Iterate on the collection
    while(iterator.hasNext()){
      Tuple nextTuple = iterator.next();
      if(nextTuple.stringLen == 0){
        //There was no match for this
character.  Just
        // append it to the StringBuffer.
        reconData.append(nextTuple.nextChar);
      }else{
        //This Tuple contains information about
a match.
        // Use that information to reconstruct
and append
        // the matching data.
        for(int cnt = 0;cnt <
nextTuple.stringLen;cnt++){
          //Iterate once for each character in
the string
          // that must be reconstructed.
          //Obtain the matching characters from
the
          // previously constructed part of the
output
          // based on an offset value and a
stringLen value
          // that are stored in the Tuple.
          char workingChar = reconData.charAt(
                    reconData.length() -
```

```
nextTuple.offset);
          //Append the character to the
StringBuffer. Note
          // that this increases the length of
the
          // StringBuffer object. Thus, the
next iteration
          // of the for loop will get the next
character
          // that is already in the
StringBuffer object.
          reconData.append(workingChar);
        }//end for loop
        //Now append the non-matching character
that is
        // stored in the Tuple.
        reconData.append(nextTuple.nextChar);
      }//end else
    }//end while

Listing 21
```

Knowing what you now know about the decompression process, no explanation beyond the comments in Listing 21 should be required.

Display the decompressed message

A copy of the original message has now been constructed. The code in Listing 22 displays the message 59 characters to the line.

```
    //Display a scale on the screen.

System.out.println("\n123456789012345678901234567890"
                          +
"123456789012345678901234567890");

    //Now display the reconstructed message 59
characters
    // to the line.
    display59(new String(reconData));

Listing 22
```

If all went according to plan, the decompressed message should be an exact replica of the original message. That is why LZ77 is referred to as a ***lossless*** compression algorithm.

Analyze the compression factor

The following analysis assumes that it is not necessary for the length of an individual binary tuple to be a multiple of eight bits. Thus, it assumes that the number of bits used to store a tuple

can be optimized on the basis of the size of the search window and the size of the lookAhead window.

The analysis also assumes that all characters require eight bits for storage and that the number of bits used to store every tuple is the same as the number of bits used to store every other tuple.

Calculate and display the compression factor

Given those assumptions, the code in Listing 23 calculates and displays the compression factor for a given message and a given pair of sizes for the search window and the lookAhead window.

```
    //Analyze the compression
    System.out.println("\nAnalyze the
compression.");
    System.out.println("Assume:");
    System.out.println(
                " 8 bits required per raw data
character.");

    //Calculate the number of bits required to
contain the
    // largest values that can occur in the
values for
    // offset and stringLen in the Tuple.
Assume that the
    // character in the Tuple is contained in 8
bits.
    int offsetBitsRequired = getBitsRequired(

searchWindowLen);
    int stringLenBitsRequired =
getBitsRequired(

lookAheadWindowLen);
    int tupleBitsRequired = 8 +
offsetBitsRequired
                                      +
stringLenBitsRequired;

    System.out.println(" Maximum offset value
of "
                       + searchWindowLen + "
char requires "
                       + offsetBitsRequired + "
bits.");
    System.out.println(" Maximum stringLen
value of "
                     + lookAheadWindowLen + "
char requires "
                     + stringLenBitsRequired + "
bits.");
    System.out.println(
                     " Character in Tuple
requires 8 bits.");
```

```
    System.out.println(" " + tupleBitsRequired
                             + " bits required
per Tuple.");
    int msgLength = rawData.length()*8;
    System.out.println("Raw data length = " +
msgLength
                                              + "
bits.");
    System.out.println("Number Tuples: "
                                          +
encodedData.size());
    int tupleLength =
encodedData.size()*tupleBitsRequired;
    System.out.println("Total Tuple length = "
                                  + tupleLength
+ " bits.");
    System.out.println("Compression factor = "
                             +
(double)msgLength/tupleLength);

  }//end doIt
```

Listing 23

Straightforward code

Although the code in Listing 23 is rather long, the only thing in Listing 23 that is even half way complicated is the invocation of the method named **getBitsRequired** to determine the number of bits required to contain the maximum possible values for **offset** and **stringLen**. The maximum possible value in each case depends on the size of the corresponding search window and lookAhead window.

The method named getBitsRequired

You can view the method named **getBitsRequired** in its entirety in Listing 24. Hopefully you will be able to understand that method without the need for additional explanation.

Formatting code

Most of the code in Listing 23 involves formatting the output that is shown in Figure 3.

The program output

Figure 3 shows the reconstructed message followed by the results of the compression analysis.

```
Miss Kalissippi from Mississippi is a cowgirl
who yells yip
pi when she rides her horse in the horse show
in Mississipp
```

```
i.

Analyze the compression.
Assume:
8 bits required per raw data character.
Maximum offset value of 31 char requires 5
bits.
Maximum stringLen value of 7 char requires 3
bits.
Character in Tuple requires 8 bits.
16 bits required per Tuple.
Raw data length = 960 bits.
Number Tuples: 53
Total Tuple length = 848 bits.
Compression factor = 1.1320754716981132
```
Figure 3

The bottom line

The bottom line in Figure 3 is that according to the assumptions described above, a binary version of the compressed message would be 1.132 times ***smaller*** than the original version of the message. Stated differently, the size of the compressed message would be 88.3 percent of the size of the original message.

Vary some parameters and rerun

Increasing the size of the search window from 31 characters to 63 characters doesn't help with the compression. In fact, the change causes the compression factor to be reduced from 1.132 to 1.107.

Increasing the size of the lookAhead window from 7 characters to 15 characters while leaving the size of the search window at 63 characters reduces the compression factor even further, from 1.107 to 1.046.

However, decreasing the size of the lookAhead window from 7 characters to 3 characters while setting the size of the search window back to 31 increases the compression factor slightly from 1.132 to 1.143.

A different message

As you may have guessed by now, the message about the cowgirl from Mississippi was designed to cause it to contain lots of repetitive substrings.

Figure 4 shows the compression results for a news story written in English and taken directly from an Internet web site. Nothing was done to cause this message to be well-suited or poorly-suited for compression using the LZ77 algorithm

```
BAGHDAD, Iraq Violence increased across Iraq
```

```
after a lull f
ollowing the Dec. 15 parliamentary elections,
with at least
two dozen people including a U.S. soldier
killed Monday in
shootings and bombings mostly targeting the
Shiite-dominat
ed security services. The Defense Ministry
director of oper
ations, Brig. Gen. Abdul Aziz Mohammed-Jassim,
blamed incre
ased violence in the past two days on
insurgents trying to
deepen the political turmoil following the
elections. The v
iolence came as three Iraqi opposition groups
threatened an
other wave of protests and civil disobedience
if allegation
s of fraud are not properly investigated.

Analyze the compression.
Assume:
8 bits required per raw data character.
Maximum offset value of 31 char requires 5
bits.
Maximum stringLen value of 7 char requires 3
bits.
Character in Tuple requires 8 bits.
16 bits required per Tuple.
Raw data length = 5048 bits.
Number Tuples: 329
Total Tuple length = 5264 bits.
Compression factor = 0.958966565349544
```
Figure 4

A net loss in compression

As you can see from Figure 4, for a search window size of 31 characters and a lookAhead window size of 7 characters, the compression factor was only 0.9589 for this news story. This means that the compressed message was larger than the original message. That's certainly not how we want a compression algorithm to behave.

Change the sizes of the sliding windows

Increasing the search window size from 31 characters to 255 characters, and decreasing the lookAhead window size from 7 characters to 3 characters increased the compression factor from 0.9589 to 1.154. At least now the factor is greater than 1 and we're no longer in the red.

Double the length of the message

Doubling the length of the message by simply concatenating the message onto itself and leaving everything else the same increased the compression factor from 1.154 to 1.206.

> *(However, I'm not sure just how meaningful that test is since the new message not only contains repetitive substrings but also contains a complete second copy of the original message.)*

Doubling the length of the message again in the same way increased the compression factor from 1.206 to 1.234.

Doubling the length of the message one more time increased the compression factor from 1.234 to 1.248.

Continuing to double the length of the message a few more times produced very small gains in the compression factor but the gain was less each time. It looked like the compression factor would top out at around 1.26.

Run the Program

I encourage you to copy the code from Listing 24 into your text editor, compile it, and execute it. Experiment with the code, making changes, and observing the results of your changes.

Create and compress your own messages. Experiment with the sizes of the search window and the lookAhead window to see how those parameters affect the compression factor for messages of your own design.

Summary

In this lesson, I showed you how to write a Java program that illustrates lossless data compression according to the Lempel-Ziv Data Compression Algorithm commonly known as LZ77. I also explained the characteristics of the algorithm that result in data compression.

What's Next?

Future lessons in this series will explain the inner workings behind several data and image compression schemes, including the following:

- Huffman data encoding
- Run-length data encoding
- GIF image compression
- JPEG image compression

Complete Program Listing

A complete listing of the program discussed in this lesson is shown in Listing 24 below.

```
/*File LZ77v01.java
Copyright 2006, R.G.Baldwin

LZ77 is the name given to a lossless data compression
algorithm published in papers by Abraham Lempel and Jacob
Ziv in 1977. This algorithm forms the basis for many LZ
variations including LZW, LZSS and others. LZ77 is known
as a dictionary encoding algorithm.

This program implements a very close approximation to the
LZ77 compression algorithm.  While it may not match that
algorithm exactly, the match is close enough to satisfy
the primary purpose of the program, which is to explain
how the LZ77 algorithm works.

This program is provided for educational purposes only. If
you use the program for any purpose, you use it at your
own risk.  The author of the program accepts no
responsibility for any damages that may result from your
use of the program.

The program was specifically designed to serve its primary
purpose of education.  No thought was given to speed,
efficiency, memory utilization, or any other factor that
would be important in a program written for production
data compression purposes.  In some cases, the program was
purposely made less efficient by using two or more
statements to accomplish a task that could be accomplished
by a single more complex statement.

Several sample messages are hard-coded into the program.
You can switch between those messages by enabling and
disabling specific messages using comments and then
recompiling the program.  You can also insert your own
message and recompile the program to use your message if
you choose to do so.

The program first encodes a message into a collection of
Tuple objects stored in an ArrayList object using a very
close approximation to the LZ77 lossless data compression
algorithm.

Then the program decodes the message by extracting the
information from the Tuple objects and using that
information to reconstruct the original message.

The program does not actually convert the encoded
(compressed) message into a bit stream, although it
wouldn't be difficult to do so.  Since the purpose of the
effort was to illustrate the behavior of the LZ77
algorithm, and not to write a production data compression
program, converting the compressed data into a bit stream
was considered to be superfluous to the purpose.  However,
the program does provide a theoretical analysis of the
```

```
amount of compression that should be achieved if the
results were converted into a bit stream.

This program was tested against the demonstration Java
program named LZ that can be downloaded from
http://www.mathcs.sjsu.edu/faculty/khuri/animation.html

Neither the source code from that program nor the source
code from any other existing Java program was consulted
during the development of the code for this program.
Although numerous technical descriptions of the LZ77
algorithm were consulted, all of the source code in this
program is the original work of this author.

The animated interactive nature of the demonstration
program mentioned above was extremely helpful in helping
this author to understand the LZ77 algorithm.

The output from this program matches the output from the
above mentioned demonstration program for 60 characters of
input data and values of 10 for searchWindowLen and
lookAheadWindowLen. (The demonstration program is limited
to an input data length of 60 characters.)

Quite a lot of output material is displayed on the screen
when this program is running.  This material is designed to
be used in a tutorial lesson to explain the behavior of
the LZ lossless data compression algorithm.

The program was tested using J2SE 5.0 and WinXP.  J2SE 5.0
or later is required due to the use of generics.
**********************************************************/
import java.util.*;

class LZ77v01{
  //Use comments to enable and disable and to select
  //between the following demonstration messages.
/*
  String rawData = "BAGHDAD, Iraq Violence increased "
  + "across Iraq after a lull following the Dec. 15 "
  + "parliamentary elections, with at least two dozen "
  + "people including a U.S. soldier killed Monday in "
  + "shootings and bombings mostly targeting the Shiite-"
  + "dominated security services. The Defense Ministry "
  + "director of operations, Brig. Gen. Abdul Aziz "
  + "Mohammed-Jassim, blamed increased violence in the "
  + "past two days on insurgents trying to deepen the "
  + "political turmoil following the elections. The "
  + "violence came as three Iraqi opposition groups "
  + "threatened another wave of protests and civil "
  + "disobedience if allegations of fraud are not "
  + "properly investigated.";
*/
  String rawData = "Miss Kalissippi from Mississippi is a "
  + "cowgirl who yells yippi when she rides her horse in "
  + "the horse show in Mississippi.";
```

```
//A collection of Tuple objects
ArrayList <Tuple>encodedData = new ArrayList<Tuple>();

//A reference to a single Tuple object
Tuple thisTuple;

//A substring that represents the search window.
String searchSubstring;

//Working variables
int matchLen;
int matchLoc;
int charCnt;
int searchWindowStart;
int lookAheadWindowEnd;

//Modify the following values to change the length of
// the search window and/or the lookAhead window.
// Optimum values are one less than even powers of 2.
int searchWindowLen = 31;
int lookAheadWindowLen = 7;
//----------------------------------------------------//

public static void main(String[] args){
  new LZ77v01().doIt();
}//end main
//----------------------------------------------------//

void doIt(){

  //Display a scale on the screen.
  System.out.println("123456789012345678901234567890"
                  + "12345678901234567890123456789");

  //Display the raw data, 59 characters to the line.
  display59(rawData);
  System.out.println();//blank line

  //Process the message, one char at a time.
  charCnt = 0;
  while(charCnt < rawData.length()){
    //Establish the beginning of the search window.
    searchWindowStart = (charCnt-searchWindowLen >=0)
                              ? charCnt-searchWindowLen
                              : 0;
    //Establish the end of the lookAhead window.
    lookAheadWindowEnd =
         (charCnt+lookAheadWindowLen < rawData.length())
                             ? charCnt+lookAheadWindowLen
                             :rawData.length();

    //Display the contents of the search window.
    System.out.print(rawData.substring(
                     searchWindowStart,charCnt) + " - ");
    //Display the contents of the lookAheadWindow on the
```

```
      // same line as the search window, separated by a
      // dash.
      System.out.print(rawData.substring(
                      charCnt,lookAheadWindowEnd) + " - ");

      //Get a substring from the search window to search
      // for a match.
      if(charCnt == 0){
        //Begin with an empty search window.
        searchSubstring = "";
      }else{
        searchSubstring = rawData.substring(
                               searchWindowStart,charCnt);
      }//end else

      //Search the search window for a match to the next
      // character in the lookAhead window.
      matchLen = 1;
      String searchTarget = rawData.substring(
                              charCnt,charCnt + matchLen);
      if(searchSubstring.indexOf(searchTarget) != -1){
        //A match was found for the one-character string.
        // See if the match extends beyond one character.
        //Limit the length of the possible match to
        // lookAheadWindowLen
        matchLen++;//Increment length of searchTarget.
        while(matchLen <= lookAheadWindowLen){
          //Keep testing and extending the length of the
          // string being tested for a match until the
          // test fails.
          searchTarget = rawData.substring(
                                  charCnt,charCnt+matchLen);
          matchLoc = searchSubstring.indexOf(searchTarget);
          //Be careful to avoid searching beyond the end
          // of the raw data.
          if((matchLoc != -1) && ((charCnt + matchLen)
                                       < rawData.length())){
            matchLen++;
          }else{
            //The matching test failed.  Break out of the
            // loop.
            break;
          }//end else
        }//end while
        //Reduce matchLen to the longest length that
        // matched.
        matchLen--;

        //We went one step too far increasing matchLen. Go
        // back and get the location in the search window
        // where the last match occurred.
        matchLoc = searchSubstring.indexOf(
              rawData.substring(charCnt,charCnt+matchLen));

        //Increase the character counter to cause the
        // outer loop to skip the matching characters.
```

```
        charCnt += matchLen;

        //Encapsulate the following information in a Tuple
        // object.
        //1. Offset distance back to the location of the
        //   match in the search window.
        //2. Length of the match.
        //3. First non-matching character in the lookAhead
        //   window

        //Calculate the offset
        int offset =
                  (charCnt < (searchWindowLen + matchLen))
                           ? charCnt - matchLoc - matchLen
                           :searchWindowLen - matchLoc;
        //Get and save the next non-matching character in
        // the lookAhead window.
        String nextChar =
                       rawData.substring(charCnt,charCnt+1);
        //Instantiate and populate the Tuple object.
        thisTuple = new Tuple(offset,matchLen,nextChar);

        //Save the Tuple object in a Collection object of
        // type ArrayList.
        encodedData.add(thisTuple);

      }else{
        //A match was not found for the next character.
        //Encapsulate the following information in a Tuple
        // object.
        //1. 0
        //2. 0
        //3. The non-matching character.
        String nextChar = rawData.substring(
                                        charCnt,charCnt+1);
        thisTuple = new Tuple(0,0,nextChar);
        encodedData.add(thisTuple);
      }//end else

      //Increment the character counter that controls the
      // outer loop.
      charCnt++;

      //Display the contents of the Tuple on the same line
      // as the search window and the lookAhead window,
      // separated by a dash.
      System.out.print(thisTuple + " - ");

      //Display the text represented by the Tuple on the
      // same line as above, separated by a dash.
      if(thisTuple.stringLen > 0){
        //The Tuple contains a pointer to a character or
        // string in the search window.  Expand and
        // display it.
        int start = charCnt - 1 - thisTuple.stringLen
                                        - thisTuple.offset;
```

```
        int end = charCnt - 1 - thisTuple.offset;
        System.out.println(rawData.substring(
                           start,end) + thisTuple.nextChar);
      }else{
        //The tuple contains a character for which no
        // match was found.  Display it.
        System.out.println(thisTuple.nextChar);
      }//end else

    }//end while loop

/*********************************************************
The original message has been encoded into a collection of
 Tuple objects where each object contains the following
 information:
1. offset points backwards to the beginning of a
   match relative to the current location. Has a
   value of 0 if there is no match for the next
   character.
2. stringLen specifies the length of the match or 0
   if there is no match for the next character.
3. nextChar is the first non-matching character
   following a match, or the next character if there
   is no match.

It would not be particularly difficult at this point
 to iterate on the collection and to produce a binary
 stream containing these three values.  For example,
 depending on the values specified for searchWindowLen
 and lookAheadWindowLen, the first two values listed above
 could be encoded into the number of bits required to
 contain the largest possible offset value and the
 largest possible stringLen value.  The character from the
 Tuple object could be encoded into eight bits.  Then,
 the set of bits required to describe the Tuple could be
 inserted into a stream of bits.

This demonstration program does not produce such a
 compressed binary file.
*********************************************************/

    //Now decode the encoded data from the Tuple objects.
    //Reconstruct and display the original message.

    //Reconstruct the message into the following
    // StringBuffer object.
    StringBuffer reconData = new StringBuffer();

    //Get an iterator on the collection of Tuple objects.
    Iterator <Tuple>iterator = encodedData.iterator();

    //Iterate on the collection
    while(iterator.hasNext()){
      Tuple nextTuple = iterator.next();
      if(nextTuple.stringLen == 0){
        //There was no match for this character.  Just
```

```
        // append it to the StringBuffer.
        reconData.append(nextTuple.nextChar);
      }else{
        //This Tuple contains information about a match.
        // Use that information to reconstruct and append
        // the matching data.
        for(int cnt = 0;cnt < nextTuple.stringLen;cnt++){
          //Iterate once for each character in the string
          // that must be reconstructed.
          //Obtain the matching characters from the
          // previously constructed part of the output
          // based on an offset value and a stringLen value
          // that are stored in the Tuple.
          char workingChar = reconData.charAt(
                    reconData.length() - nextTuple.offset);
          //Append the character to the StringBuffer. Note
          // that this increases the length of the
          // StringBuffer object. Thus, the next iteration
          // of the for loop will get the next character
          // that is already in the StringBuffer object.
          reconData.append(workingChar);
        }//end for loop
        //Now append the non-matching character that is
        // stored in the Tuple.
        reconData.append(nextTuple.nextChar);
      }//end else
    }//end while

    //The original message has been reconstructed.

    //Display a scale on the screen.
    System.out.println("\n123456789012345678901234567890"
                       + "12345678901234567890123456789");

    //Now display the reconstructed message 59 characters
    // to the line.
    display59(new String(reconData));

    //Analyze the compression
    System.out.println("\nAnalyze the compression.");
    System.out.println("Assume:");
    System.out.println(
              " 8 bits required per raw data character.");

    //Calculate the number of bits required to contain the
    // largest values that can occur in the values for
    // offset and stringLen in the Tuple.  Assume that the
    // character in the Tuple is contained in 8 bits.
    int offsetBitsRequired = getBitsRequired(
                                          searchWindowLen);
    int stringLenBitsRequired = getBitsRequired(
                                       lookAheadWindowLen);
    int tupleBitsRequired = 8 + offsetBitsRequired
                                 + stringLenBitsRequired;

    System.out.println(" Maximum offset value of "
```

```
                    + searchWindowLen + " char requires "
                    + offsetBitsRequired + " bits.");
  System.out.println(" Maximum stringLen value of "
                  + lookAheadWindowLen + " char requires "
                  + stringLenBitsRequired + " bits.");
  System.out.println(
                  " Character in Tuple requires 8 bits.");

  System.out.println(" " + tupleBitsRequired
                         + " bits required per Tuple.");
  int msgLength = rawData.length()*8;
  System.out.println("Raw data length = " + msgLength
                                          + " bits.");
  System.out.println("Number Tuples: "
                                    + encodedData.size());
  int tupleLength = encodedData.size()*tupleBitsRequired;
  System.out.println("Total Tuple length = "
                                 + tupleLength + " bits.");
  System.out.println("Compression factor = "
                       + (double)msgLength/tupleLength);

}//end doIt
//-----------------------------------------------------//

//Method to display a String 59 characters to the line.
void display59(String data){
  for(int cnt = 0;cnt < data.length();cnt += 59){
    if((cnt + 59) < data.length()){
      //Display 59 characters.
      System.out.println(data.substring(cnt,cnt+59));
    }else{
      //Display the final line, which may be short.
      System.out.println(data.substring(cnt));
    }//end else
  }//end for loop
}//end display59
//-----------------------------------------------------//

//Method to return the number of bits required to
// contain a given integer value.  (There must be a
// better way to do this, such as finding the log to the
// base 2.)
int getBitsRequired(int value){
  if(value < 2){
    return 1;
  }else if(value < 4){
    return 2;
  }else if(value < 8){
    return 3;
  }else if(value < 16){
    return 4;
  }else if(value < 32){
    return 5;
  }else if(value < 64){
    return 6;
  }else if(value < 128){
```

```
      return 7;
    }else if(value < 256){
      return 8;
    }else{
      System.out.println(
         "Bit conversion too large, terminating program.");
      System.exit(1);
      return 0;//Make compiler happy.
    }//end else
  }//end getBitsRequired
  //-----------------------------------------------------//

  //The purpose of this inner class is to provide a
  // wrapper for the three critical data values that
  // are produced during implementation of the LZ77
  // compression algorithm.
  class Tuple{
    //offset points to the beginning of a match relative
    // to the current location.  Has a value of 0 if there
    // is no match.
    int offset;
    //stringLen specifies the length of the match or 0 if
    // there is no match
    int stringLen;
    //nextChar is the first non-matching character
    // following the match, or the only character if there
    // is no match.
    String nextChar;
    //-----------------------------------------------------//

    //Constructor
    Tuple(int offset,int stringLen,String nextChar){
      this.offset = offset;
      this.stringLen = stringLen;
      this.nextChar = nextChar;
    }//end constructor
    //-----------------------------------------------------//

    //Overridden toString method
    public String toString(){
      return offset + "," + stringLen + "," + nextChar;
    }//end toString
  }//end class Tuple
  //-----------------------------------------------------//
}//end class LZ77v01
```

Listing 24

About the author

Richard Baldwin *is a college professor (at Austin Community College in Austin, TX) and private consultant whose primary focus is a combination of Java, C#, and XML. In addition to the many platform and/or language independent benefits of Java and C# applications, he believes that a combination of Java, C#, and XML will become the primary driving force in the delivery of structured information on the Web.*

Richard has participated in numerous consulting projects and he frequently provides onsite training at the high-tech companies located in and around Austin, Texas. He is the author of Baldwin's Programming Tutorials, which have gained a worldwide following among experienced and aspiring programmers. He has also published articles in JavaPro magazine.

In addition to his programming expertise, Richard has many years of practical experience in Digital Signal Processing (DSP). His first job after he earned his Bachelor's degree was doing DSP in the Seismic Research Department of Texas Instruments. (TI is still a world leader in DSP.) In the following years, he applied his programming and DSP expertise to other interesting areas including sonar and underwater acoustics.

Richard holds an MSEE degree from Southern Methodist University and has many years of experience in the application of computer technology to real-world problems.

Baldwin@DickBaldwin.com

Keywords
java Lempel Ziv LZ77 lossless compression algorithm LZW LZSS DEFLATE tuple

-end-