Chapter 2 - Compression

As one might expect, the continuous aspect of multimedia data can require significant resources from the systems that support it. Consider an HDTV quality video stream with the following parameters: 1920x1080 pixel frames, 30 frames per second, and 3 bytes per pixel (red, green, blue). In order to represent such data, the following bit requirement is necessary:

$$\left(\frac{1920 * 1080\ pixels}{frame}\right)\left(\frac{3bytes}{pixel}\right)\left(\frac{8bits}{byte}\right)\left(\frac{30\ frames}{\text{sec.}}\right) = 1{,}492{,}992{,}000 \quad bits / \text{sec.}$$

1.5 Gigabits per second or 177 megabytes per second is quite a significant load for most computing and storage systems. For example, in order to store a two hour movie would require approximately 54 Blu-Ray discs to hold the entire movie! Clearly changing a disc every 2.5 minutes would not lead to a pleasant user experience. For multimedia systems, the compression of data makes the transmission and storage of media more scalable.

2.1 Compression Overview

2.1.1 Basic Definitions

Compression is the process of altering the representation of uncompressed data into another form that is, hopefully, much smaller in size. *Decompression* reverses the compression step and results in the original representation of the data (or a near equivalent). There are two types of compression that can be employed to reduce the amount of data for a signal, *lossless* and *lossy* data compression.

Lossless data compression is *reversible*. That is, the exact original data is returned after the decompression algorithm has been run on the compressed data. Lossless compression is sometimes referred to as *entropy encoding*. Lossless compression techniques take advantage of the fact that in many cases there is redundant information within the data. Compressing a random sequence of 8-bit data symbols will typically not result in any compression. Examples of lossless compression algorithms include Huffman compression, arithmetic coding, run length encoding, and Lempel Ziv Welch compression.

Lossy data compression results in the *removal* of some data from the original uncompressed data. This process can either (i) reduce or abstract the data or (ii) take advantage of some physical or human limitation in the representation of the data. As an example of reducing or abstracting the data set, suppose we have a picture that is 2000 pixels wide and 1000 pixels tall. One could represent the image as a 1000x500 pixel image, reducing the overall amount of data by 75%. As an example of the latter, humans can generally hear frequencies up to about 20,000 Hz while other "signals" such as dog whistles may exist that they cannot hear. As a result, frequencies well above this can be removed without any *perceived loss to the human*. We will describe lossy data compression techniques that take advantage of human limitations in the next chapter.

2.1.2 Implementing Compression Algorithms

Compression and decompression algorithms work together to provide users with the reduction in data storage or transmission that they are looking for. From a user's perspective algorithms such as *gzip* or *gunzip* are relatively easy to use: run *gzip* to compress the file and run *gunzip* to uncompress the file. In implementing these or any other compression/decompression algorithms, however, a number of concerns and trade-offs need to be addressed.

First, *both compressor and decompressor need to agree on a common format* that will be used to represent the compressed stream. This includes several components including:

- a magic cookie – the magic cookie allows the decompressor program to identify the type of file. For example, all PNG image files begin with the following 8 byte hexadecimal sequence: 89 50 4e 47 0d 0a 1a 0a. In ASCII, this translates to the following 8 characters: \211 P N G \r \n \032 \n.
- a header – the header is used by the compressor to specify various types of parameters that were used in compression. These parameters allow the decompressor to use the appropriate options in the decompression step. For example, some algorithms require the inclusion of a decoding dictionary, which would normally be specified in the header of the file.
- the compressed data – typically the rest of the file will represent the compressed data.

Second, the compressor and decompressor must agree upon the exact algorithm to be used. This may include both an implicitly agreed upon algorithm or an explicitly declared algorithm. As an example of the implicitly agreed upon format, consider the following text message exchange over a cell phone:

AAR, C u l8r 2nite

For some text users, this translates to "At any rate, see you later tonight". If you happen to text a lot, implicit in the "dictionary" of the sender and the receiver are the translations of acronyms and shorthand representations into their respective meanings. Now consider a person (decompressor) that does not know the dictionary. In order for the person to decompress the message, either the dictionary needs to be well known (implicit) or the dictionary needs to be transmitted with the text (explicit). Obviously, for text messaging, transmitting the dictionary with the text defeats its entire purpose. The most important aspect of all this when it comes to compressing and decompressing data is that all such knowledge needs to be built into the algorithms from the beginning.

With the agreed upon format and algorithm, the last aspect to consider is the *compression overhead*. In order to compress a file, the various compression algorithms introduce some compression overhead. That is, in order to represent the compressed file, the header typically expands the size of the file in order to set up the compression algorithm, possibly sending a dictionary as part of it.

2.1.3 Measuring Compression Performance

In order to compare the performance of various compression algorithms or to measure the performance of a single compression algorithm, a way to measure its impact is necessary. The common method is to determine how much smaller the compressed file size is compared

to the original file size for a variety of expected use cases. This number is reported as the *compression ratio* which is defined as:

$$compression\ ratio = \frac{size\ of\ uncompressed\ file}{size\ of\ compressed\ file}$$

The resulting compression ratio, *x*, is typically reported as "*x* to 1 compression ratio". As an example, consider a text file that is 1400 bytes in size. Further, suppose the compressed file is 700 bytes. The resultant compression ratio is then 2:1 or is said to have achieved a "2 to 1 compression ratio". Obviously, this just means the compressed file is half the size of the original file.

2.2 Lossless Data Compression Algorithm

In the rest of this subsection, we will describe several useful compression algorithms. The treatment in this book is not intended to be complete. A reference for data compression in general is provided in the References section at the end of the chapter.

Generally, compression algorithms take input *symbols* and translate them into compressed *codewords*, where the codewords should theoretically be smaller than the symbol(s) that they represent. Most lossless compression algorithms are able to achieve approximately 2-4:1 compression ratios.

2.2.1 Huffman Compression

Huffman compression was invented in 1952 by David Huffman as a way to optimally assign an unambiguous encoding to an input set of symbols. The algorithm is a two-pass algorithm that assigns *variable-length* codewords to *fixed-length* input symbols. The first pass gathers statistics regarding the frequency of the individual input symbols. The second pass assigns the codewords to the input symbols. The size of the variable-length codewords is inversely proportional to the input symbol probability in the data set. The size of the codewords is approximately governed by:

$$\#\ bits \cong \log_2(symbol\ probability)$$

2.2.1.1 Compression with Huffman

To create the optimal assignment of codewords to input symbols, the Huffman compression algorithm creates a *Huffman Tree* that represents the individual bits within a codeword that will be used. The Huffman Tree is created by assigning the input frequency to a symbol or a sub-tree of symbols. When all input symbols have been assigned (or connected) to the tree, the optimal assignment is complete. The pseudo-code for the optimal Huffman Compression algorithm is shown in Figure 2.1 (on the next page).

```
Given: Characters with either distribution or number of
occurrences

1) Find smallest two values
2) Combine into a node
          Mark new node with combined dist.
          Assign 0 and 1 to branches
3) If all combined into a single tree, goto 4,
     otherwise go to step 1


4) Starting from the root, labels on the branches to a
     particular symbol make up the codeword
```

Figure 2.1 Huffman Pseudo-code

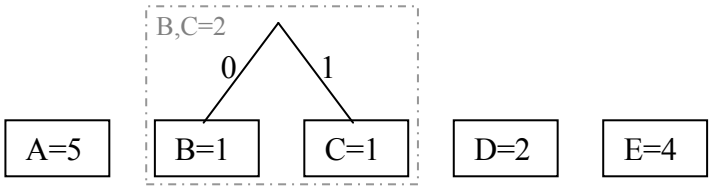
As an example, consider the following input file:

A A A A A B C D D E E E E

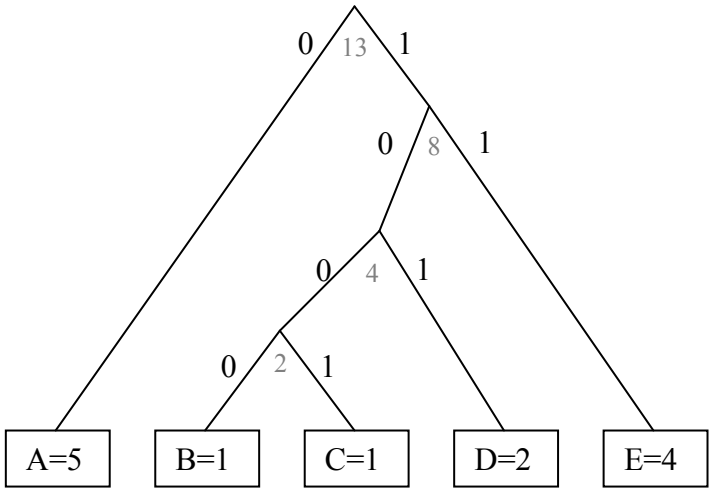
In the first pass, we determine the input symbol frequency. Here we end up with the following table:

Symbol	# occurrences	probability
A	5	38%
B	1	8%
C	1	8%
D	2	15%
E	4	31%

To create the Huffman Tree, the algorithm takes the input symbols and creates a subtree from the two smallest probability sub-trees or input symbols and assigns a 1 and 0 to each of the links to the sub-trees or input symbols. For the above example, the first step combines the two lowest probability symbols B and C into a subtree (since the number of occurrences is proportional to the symbol probability, we will use the symbol occurrence in the diagrams:



The second step then combines the next two smallest sub-trees or symbol frequencies. Here, the combined subtree has overall occurrence of 2 and is combined with D. The entire Huffman Tree for this example is shown below:



The 0's and 1's in the tree represent the individual bits within the codewords. The actual bits in the codeword are determined by traversing from the top (root) of the tree to the individual symbols. Thus, to compress "B", the algorithm outputs the following codeword: 1000. The complete dictionary for this example is:

Symbol	Huffman String
A	0
B	1000
C	1001
D	101
E	11

Other equivalent assignments are possible (e.g. switching all 1's for 0's or switching the 1 and 0 assignments in the middle of the tree). One thing to note, however, is that each prefix is unique for each of the symbols, allowing decompression to be unambiguous.

In order to calculate the compression ratio for this example, we first need to determine the uncompressed data size for the file. Assuming 8-bit ASCII characters, the uncompressed size is 8*13 bits or 104 bits. The compressed file size can be calculated (start from A to E) as: 5*1 + 1*4 + 1*4 + 2*3 + 4*2 = 27 bits. Thus, the compression ratio is 104/27 or 3.85:1. In general, *however*, the compression ratio will be much smaller (i.e, not as good). The reason for this is that the actual optimal symbol dictionary will not be known by the decompressor. There are two options (i) have the compressor and decompressor agree on what the table ought to be or (ii) include the symbol dictionary in the compressed file. There are pros and cons to each of the above approaches. If the compressor and decompressor agree on a common table ahead of time (e.g. the distribution of characters in the English language), then the table will not be optimal. However, the table does not need to appear in the compressed data file. If the table needs to be included in the file, then the overall compression of the file will decrease. As an example, consider the working example above. In order to specify the table, 3-bytes can be used for each entry, one for the symbol, one for the length of the codeword, and the actual codeword. Thus the overall compression ratio would actually be:

$$Compression\ ratio = \frac{104\ bits}{3*5*8\ bits + 27\ bits} = \frac{104\ bits}{147\ bits} = 0.70$$

What this means is that including the table for this small example requires more space than the original file itself! The key here is that there needs to be a large enough sample of data in order to amortize the cost of the table in the compressed stream.

2.2.1.2 Decompression with Huffman

To decompress a Huffman encoded file, the process starts by reading the bits in the compressed data and finds the longest possible prefix match in the codeword dictionary. Suppose we have the following compressed sequence:

```
100101011010...
```

In this case, the first bit (1) matches B, C, D, and E. The first two bits (10) match B,C, and D. The first three bits (100) match B and C. The first four bits (1001) match C only. Thus, the first symbol in the compressed sequence is C.

```
1001011010
```

Starting at the fifth bit, 0 matches A so the second symbol is A. Continuing the decompression, the resultant uncompressed data is:

```
100101011010
C   AD  D  A
```

Decompression using Huffman is typically quite time consuming as the compressed stream is parsed at the bit level rather than the byte level.

Huffman decompression is typically implemented with a speed up table that allows codewords to be more quickly converted into the original symbol. The key to doing this is that each of the prefixes in a Huffman tree are unique and therefore can be used in an array look-up table. Consider the previous example where the codebook contains:

Symbol	Huffman String
A	0
B	1000
C	1001
D	101
E	11

Since the longest Huffman string is 4-bits in length, we can populate a 16-entry array that is indexed by the Huffman string *extended* to all possible 4-bit entries covered by the Huffman String prefix. The above table would then be represented as:

Array Index	Symbol	#bits in Huffman String
0 (0000)	A	1
1 (0001)	A	1
2 (0010)	A	1
3 (0011)	A	1
4 (0100)	A	1
5 (0101)	A	1
6 (0110)	A	1
7 (0111)	A	1
8 (1000)	B	4
9 (1001)	C	4
10 (1010)	D	3
11 (1011)	D	3
12 (1100)	E	2
13 (1101)	E	2
14 (1110)	E	2
15 (1111)	E	2

The decompression process is then accomplished as follows. The decompressor takes 4 bits of the input at a time. For example, suppose we have the following sequence:

```
1010111000
```

We take the first four bits (1010) and use this as the index into the look-up table. In entry 10, we see D with the number of bits being 3. This means that we "consume" the 101 and get three more bits for the looking up the next symbol:

```
101<u>0111</u>000
D
```

Looking up (0111), we get A with bit length 1. We consume 1 bit and get another bit from the input:

```
1010<u>1110</u>00
D  A
```

(1111) leads to E with code length of 2 bits:

```
101011<u>1000</u>
D  AE
```

Finally, (1000) yields B:

```
1010111000
D  AE B
```

The important thing to note is that for each codeword converted back to its original symbol, the number of bits in the input that are consumed is equal to its codeword length. This greatly simplifies the number of bit operations required to convert codewords back into their original symbols.

2.3 Run Length Encoding

Run length encoding (RLE) is a lossless compression algorithm that records the number of repeated occurrences of an input symbol. There are many ways in which RLE can be implemented, each with their own semantics for what a "run" constitutes.

Basic RLE

In its simplest form, RLE takes an input string of symbols and replaces each set of recurring input symbols (possibly only one) with the tuple <input symbol, # of occurrences>. For example, suppose the input sequence is as follows:

```
A A A A B B B B B C D D
```

The sequence is then replaced with:

```
A [4] B [5] C [1] D [2]
```

where [i] represents the binary digit *i*. The compression ratio for this small example is:

$$Compression\ ratio = \frac{12}{8} = 1.5$$

The compression ratio of RLE depends entirely upon the length of repeated input symbols. Obviously, in the English language the number of occurrences of repeated letters is typically no more than two so compressing text with RLE will not result in any compression. RLE has a number of applications where it can be useful, however. These include:

- Graphical text images, where characters are repeated many times to represent a shade of gray.
- Data that has been manipulated in order to create large runs of the same input symbols. As we will see later on, JPEG uses RLE in its entropy encoding.

RLE Variations

There are several variations of RLE compression that can be employed.

Variation 1: First, for some data sets that have many runs of length one, RLE performs poorly because each singleton input symbols requires *two* output bytes. To get around this, one can only encode runs of length 2 or more. Singleton input symbols are coded with just the symbol. If it is more than one, the input symbol appears twice in the output sequence appended with the run length minus one. As an example:

```
A B C C C D E F F F F F G
```

is encoded as:

```
A B C C [2] D E F F [4] G
```

Here, the recurrence of the character denotes to the decompression algorithm that a run is going to follow. By consistently following this encoding scheme, the decompression algorithm can unambiguously recover the original data.

Variation 2: A second variation is to use *escape sequences* to denote the beginning of runs and to only encode runs greater than 1. For example, suppose we have the following sequence:

```
A B C C C D E F F F F F G
```

We can use the * character to indicate a run and follow it with the run length. For the above example, we get:

```
A B C * [3] D E F * [5] G
```

Variation 3: A third variation that can be used is when the data has been manipulated in order to create a large number of runs of the same input symbol. For example, suppose we have the following sequence from a sensor:

```
2 3 5 6 7 8 9 10 14 15 16 17 18
```

Assuming the application typically sees such monotonically increasing sequences. The compression algorithm could perform differential compression, where the coded sequence is represented as differences (after the first symbol). Thus, the sequence turns into:

```
2 1 2 1 1 1 1 1 4 1 1 1 1
```

That is, the difference between 2 and 3 is 1, 3 and 5 is 2, and so on. Now, in examining this sequence, there are a large number of 1's present. One could assume that the run being encoded is the number of 1's that occur *before the next symbol.* Thus, we can turn the above 2 1 2 1 1… sequence into:

```
2 ([2], 2) ([6], 4) ([4], 1)
```

or

```
2 [2] 2 [6] 4 [4] 1
```

RLE Implementation Issues

There are a number of minor implementation issues involved with RLE compression. In *variation 2*, a special escape character "*" was used to signal the start of a run encoding. This character, however, might actually occur as part of the input sequence. As an example, suppose we are using *variation 2* and have the following input sequence:

```
A B C C C D * F F F F F
```

In order to get around this, we can simply use the escape character twice to denote the * input symbol. Thus, using *variation 2* RLE encoding, we get the following:

```
A B C * [3] D * * F * [5] G
```

Another problem that arises in RLE compression is that the run might be greater than 255. Thus, it may be too long to fit within one output byte. To get around this problem, we can simply break the run into two smaller runs, with a maximum run of 255.

2.4 Lempel Ziv Compression

As we have seen already, Huffman compression takes a fixed size input symbol size (typically one byte) and converts it into a variable sized output codeword whose size is inversely proportional to the percentage of times that it occurs in the input sequence. Lempel Ziv (LZ) compression takes a variable number of input symbols and creates fixed-size codewords. There are a number of goals that LZ compression is trying to accomplish. First, in order to achieve the best compression Huffman compression requires two passes over the input data. LZ adaptively discovers recurring sequences in the input data, requiring only one pass over the data. This makes it amenable for use as an inline compression algorithm; for example, in use with a data networking protocol. Second, because the dictionary that converts a number of input symbols into the output codewords is dynamically created, no table needs to be transmitted with or stored with the compressed data. The only drawback of LZ compression is that it requires conditioning (creating) of the dictionary. Thus, little

compression is achieved initially. Once the table is sufficiently large, however, compression becomes more effective.

LZ (and its variants such as LZW) are used in a number of applications such as (i) the UNIX programs *compress*, *gzip, gunzip*, (ii) network data compression (e.g. v.42 modem compression, and (iii) GIF, PNG, and TIFF image compression.

LZ Compression

The basic LZ compression algorithm is shown below

```
1.  w = nil
2.  while (read char k)
3.    if wk exists in dictionary  // extend string
4.       w = wk
5.    else                        // add new dictionary entry
6.       add wk to dictionary
7.       output code for w
8.       w = k
9.  output code for w             // output last symbol
```

Line 1 initializes *w* to nil. The purpose of *w* in the code is to hold sets of input symbols that have been seen before. If the set of input symbols *wk* is in the dictionary (line 3), then the algorithm attempts to extend *w*, reading in more characters until a longer set of continuous input symbols has not been seen. If the input symbol *wk* has not been seen before (line 5), we will add it to the dictionary (line 6) for look up later and output the codeword for *w*, which may come out of the dictionary.

LZ is best understood through an example. Suppose we have the following input sequence:

```
A B C D A B C A B B A B B D A B C E
```

For now, assume that the dictionary has 512 entries in it. The first 256 entries contain the entries for the input symbols 0-255. The entries from 256 to 511 are use to hold the strings that are learned. For the first pass, the program takes A and discovers it exists in the dictionary (the binary ASCII number for the letter "A"). It then goes back to line 2 and reads in B. At this point, the string AB does not exist in the dictionary so it is added at entry 256. The codeword for *w* (which is "A") at this point is output. The algorithm then assigns *w* to B and continues. Thus, the program trace for the above looks as follows:

```
w       k          dictionary entries        output
A       B          <256>    AB                 A
```

Continuing the example, *w* is now equal to B so we have the following:

```
w       k          dictionary entries        output
A       B          <256>    AB                 A
B
```

We then return to line 2 and read in C.

```
w       k          dictionary entries        output
A       B          <256>    AB                 A
B       C
```

We then return to line 2 and read in C. The *wk* sequence BC has not been seen before so it is added to the dictionary and the output symbol for B is output.

w	*k*	*dictionary entries*		*output*
A	B	<256>	AB	A
B	C	<257>	BC	B

It is important to note that *ALL OUTPUT codewords are 9-bits in length!* Thus, the A and B shown above are actually start with an extra 0 bit. Continuing the example:

A B C D A B C A B B A B B D A B C E

w	*k*	*dictionary entries*		*output*
A	B	<256>	AB	A
B	C	<257>	BC	B
C	D	<258>	CD	C
D	A	<259>	DA	D
A	B	exists!		

At this point, *w*=A and *k*=B has just been read in. *wk* exists in the dictionary so *w* now becomes "AB". Continuing, C is then read in and ABC is added to the dictionary:

A B C D A B C A B B A B B D A B C E

w	*k*	*dictionary entries*		*output*
A	B	<256>	AB	A
B	C	<257>	BC	B
C	D	<258>	CD	C
D	A	<259>	DA	D
A	B	exists!		
AB	C	<260>	ABC	<256>

In order to output *w*, which is AB, we output the binary sequence <256>, also a 9-bit number. Thus, we have now represented two input symbols with one 9-bit binary output codeword.

A B C D A B C A B B A B B D A B C E

w	*k*	*dictionary entries*		*output*
A	B	<256>	AB	A
B	C	<257>	BC	B
C	D	<258>	CD	C
D	A	<259>	DA	D
A	B	exists!		
AB	C	<260>	ABC	<256>
C	A	<261>	CA	C
A	B	exists!		
AB	B	<262>	ABB	<256>
B	A	<263>	BA	B
A	B	exists!		
AB	B	exists!		
ABB	D	<264>	ABBD	<262>
D	A	exists!		
DA	B	<265>	DAB	<259>
B	C	exists!		
BC	E	<266>	BCE	<257>
E	<EOF>			E

Thus, the overall compressed stream is as follows:

A B C D <256> C <256> B <262> <259> <257> E

As demonstrated by the example, initially, little compression takes place; the input symbols are replicated to the output with an additional 0 bit prepended to make them 9-bits. Once several occurrences of the same sequence of letters appears, the algorithms starts to output entries that are combinations of input sequences. For text files, one can image dictionary entries for words such as "the " occurring quite quickly. The overall compression for the above example can be calculated as:

$$Compression\ ratio = \frac{18 * 8}{12 * 9} = \frac{144}{108} = 1.3$$

The most important aspect of LZ compression is that there is no explicit dictionary that needs to be stored with the file. The first several entries in the compressed stream allow the decompressor to build the dictionary on the fly.

LZ Decompression

The basic LZ decompression algorithm is shown below:

```
1.  read k;
2.  output k;
3.  w = k;
4.  while (read char k)
5.    entry = dictionary entry for k
6.    output entry
7.    add w + entry[0] to dictionary
8.    w = entry
```

Obviously, the purpose of this algorithm is the reconstruction of the original data. On line 5, *entry* is assigned the dictionary contents for *k*. In line 7, *entry[0]* refers to just the first symbol in the *entry* string.

The first 3 lines are used to initialize the variables. Lines 4-8 do the rest of the work. For the previous example for LZ compression, we have:

```
A B C D <256> C <256> B <262> <259> <257> E
```

After the first 7 lines, we have the following program trace:

w	*k*	*entry*	dictionary	output
A		A		A
A	B	B		B

In line 7, we add the first real dictionary entry. Here *w* is "A" and *entry* is "B". Thus, *entry[0]* is "B" and "AB" is added to the dictionary at position 256.

w	*k*	*entry*	dictionary	output
A		A		A
A	B	B	<256> AB	B

Continuing, we get the following output:

```
A B C D <256> C <256> B <262> <259> <257> E
```

w	*k*	*entry*	dictionary	output
A		A		A
A	B	B	<256> AB	B
B	C	C	<257> BC	C
C	D	D	<258> CD	D
D	<256>	AB	<259> DA	AB

Here "DA" is added to the dictionary because the *entry* is AB and *entry[0]* is "A". The overall program trace is then:

```
A B C D <256> C <256> B <262> <259> <257> E
w       k      entry          dictionary             output
A               A                                     A
A       B       B            <256>  AB                B
B       C       C            <257>  BC                C
C       D       D            <258>  CD                D
D     <256>     AB           <259>  DA                AB
AB      C       C            <260>  ABC               C
C     <256>     AB           <261>  CA                AB
AB      B       B            <262>  ABB               B
B     <262>     ABB          <263>  BA                ABB
ABB   <259>     DA           <264>  ABBD              DA
DA    <257>     BC           <265>  DAB               BC
BC      E       E            <266>  BCE               E
```

Note, the original input sequence to the compressor has been retrieved. Also, note that the final dictionary is *exactly* the same as in the compressor.

The KwKwK sequence

As previously mentioned, LZ is adaptive and builds the dictionary necessary for decompression on-the-fly. There is one case, however, that can cause a small problem for the decompressor. Suppose we have the following input sequence for compression:

```
A B A B A B A B A
```

Doing the compression for the sequence we end up with the following program trace:

```
A B A B A B A B A
w       k          dictionary entries        output
A       B          <256>    AB                 A
B       A          <257>    BA                 B
A       B           exists
AB      A          <258>    ABA                <256>
A       B           exists
AB      A           exists
ABA     B          <259>    ABAB               <258>
B       A           exists
BA     <EOF>                                   <257>
```

Running the decompression algorithm, we end up with the following program trace:

```
A B <256> <258> <257>
w       k      entry          dictionary             output
A               A                                     A
A       B       B            <256>  AB                B
B     <256>     AB           <257>  BA                AB
AB    <258>   Oops!!!!
```

As shown, in the decompression, we end up in a situation where we need the dictionary entry for <258>, however, <258> has not yet been defined. This is known as the *KwKwK* sequence, where *K* is a single symbol and *w* is a dictionary entry or single symbol. If *KwKwK* exists in the input sequence and *Kw* exists in the dictionary before the *KwKwK* starts

to be compressed, then we end up in the situation where the dictionary entry needed for decompression has not yet been defined. All is not lost, however.

An observation is that entry <258> will consist of "AB" (which is *w* in this case) + *entry[0]*. In a somewhat recursive manner, we now know that the first letter (obtained from the "AB" string) of *entry[0]* is, in fact, "A". Thus, we know that <258> will actually be *w+entry[0]*, which is "AB" + "A", or "ABA". We can then enter this in the dictionary:

```
A B <256> <258> <257>
w       k      entry          dictionary             output
A                A                                      A
A       B        B            <256>  AB                 B
B     <256>      AB           <257>  BA                 AB
AB    <258>                   <258>  ABA
```

Having this, we can then continue on with the decompression:

```
A B <256> <258> <257>
w       k      entry          dictionary             output
A                A                                      A
A       B        B            <256>  AB                 B
B     <256>      AB           <257>  BA                 AB
AB    <258>      ABA          <258>  ABA                ABA
ABA   <257>      BA           <259>  ABAB               BA
```

Other LZ Implementation Issues

There are a number of issues that arise when implementing LZ compression. First, in the previous examples, we have assumed that the dictionary has 512 entries: 256 for the possible input symbols and 256 for dictionary entries that combine input symbols. For a large enough sequence, the dictionary will eventually get full. There are a number of options at this point.

Option 1: Keep the dictionary fixed once full. When all the entries are used up, the LZ algorithm simply looks up the longest possible strings and outputs the dictionary codeword for it. This has the advantage of a fixed memory size dictionary and makes the implementation simpler. The disadvantage is that it may not achieve as high a compression ratio as possible.

Option 2: Grow the dictionary by adding another bit to the output codewords. For the examples above, once the dictionary entry for <511> has been used, then the compressor fixes the first 512 entries, growing the dictionary by one bit and adding another 512 undefined entries to the dictionary. Note, the compressor and decompressor have to change to 10-bit codewords at the same time.

Option 3: Grow the dictionary until a certain number of entries and then fix the dictionary. Here, the dictionary could grow until a fixed number of entries have been defined. For example, one could grow the codewords to a maximum of 12-bits. Once all 2^12 entries have been defined, then the dictionary is fixed from then on.

Option 4: Manage the dictionary entries explicitly. Once the dictionary becomes full, the compressor and decompressor can agree on a common way to reduce the number of entries in the dictionary. For example, once the dictionary becomes full, the compressor and decompressor might decide to remove half of the dictionary entries that were least used. This

requires that the compressor and decompressor keep track of which dictionary entries have been used and manage the dictionary in exactly the same way.

Option 5: Clear out all the dictionary entries. Another possibility is to explicitly flush the dictionary. Commonly, the dictionary entry <256> is reserved for such purposes. If, in the decompression step, the decompressor finds <256> this is used as a signal from the compressor to the decompressor that it should clear out the dictionary.

Other options are possible and combinations of the above options are also possible. The most important thing is to keep the compressor and decompressor synchronized in how they are managing the dictionaries.

2.5 Chapter Summary

In this chapter, we have covered a number of basic data compression (entropy encoding) techniques including Huffman, Run-Length-Encoding, and Lempel Ziv-based compression. Some important concepts to remember for compression in general are:

- The compressor and decompressor need to agree on the format of what a valid compressed file is and how to specify various options in the header of the file.
- Except for dynamically built dictionary algorithms, the dictionary of how to decompress a file needs to be included within the compressed stream or must be a commonly agreed upon dictionary (e.g. the distribution of English characters). The common dictionary may be specified by the standard or a document describing its implementation.
- Compression algorithms take advantage of the redundancy available within the file. Without redundancy, the compression ratio will remain 1 or in some circumstances (e.g. LZ) may result in a larger file.

As we begin the discussion of audio, images, and video, we will see that these techniques help in the compression of the multimedia data.

2.6 Problems

1. a) Compress the following sequence of characters using an optimal Huffman compression codeword assignment:

   AAAAAAABCCCDDDDEEEEEFFGGGG

   b) Assuming an infinite file sequence, what is the compression ratio achieved by the optimal codeword assignment

   c) For part (a), assuming the optimal table is encoded with 3-bytes per entry into the compressed file, what size file (assuming the same distribution of characters) is necessary in order to achieve a 2:1 compression ratio?

2. a) Suppose we have a language that we have been gather the statistical distribution of letters for. They are as follows:

   A 0.20

B 0.40
C 0.15
D 0.05
E 0.20

b) For part 1, assuming the optimal table is encoded with 3-bytes per entry into the compressed file, what size file is necessary in order to achieve a 2:1 compression ratio?

3. In the run-length encoding section, we described two possible methods for encoding run-length for text. Suppose we have the following sequence:
   AAAAAAABBBBBCCCDDDDEEEEEFFFFFFGGGGHHI
   Which encoding is smallest?

4. Compress the following sequence using LZW compression
   A_BAT_A_CAT_A_HAT_A_MAT
   Show your dictionary entries and calculate the compression ratio.

5. Compress the following sequence using LZW compression
   AAAAAAAABABAABAB
   Show your dictionary entries and calculate the compression ratio.

6. Decompress the following sequence using LZW decompression. Assume an encoding similar to that provided in the LZW discussion.
   F O R _ <256> R E <259> O U <258> <260>
   Show your dictionary entries.

7. Decompress the following sequence using LZW decompression. Assume an encoding similar to that provided in the LZW discussion.
   F A L <257> <259> <258> B A H
   Show your dictionary entries.

8. Suppose we have three LZ compression algorithms. Algorithm 1 grows the dictionary to 512 entries and then fixes the dictionary for the rest of the time. Algorithms 2 grows the dictionary to 512 entries and then removes 256 of the least used entries (and continues to do so for the entire file). Algorithm 3 grows the dictionary to 512 entries and then throws out the dictionary, starting a new dictionary. Under what conditions would you expect each of the algorithms to work the best? Hint: consider the type of content, the amount of computation, etc..

   Which one would you choose to use for all types of files? Why?

2.7 References

K. Sayood, *Introduction to Data Compression*, Morgan Kaufmann, ISBN 1558605584, 2000.