A new compression and decompression algorithm for files and packets

James A. Riechel

November 30, 2007

Abstract

A new algorithm for compressing and decompressing data is presented which is not computationally expensive, produces interesting partially encoded and non-encoded output, performs well on all types and different types of input, and which might be appropriate for the compression and decompression of files, as well as packets traveling across a network. The algorithm requires computation on the order of $O(n)$, where n specifies the linear size of the input. We make a logical argument that the time complexity of both algorithms is $O(n)$, and empirical evidence supports running times of $O(n)$ for both the compression and the decompression algorithms. We consider the case of compressing and decompressing completely random data, using a random number generator which we will not describe. Both the compression and decompression algorithms require $O(n)$ time to compress and to decompress completely random data, and the output of the compression algorithm is not much larger than the completely random input.

1 Compression algorithm

The compression algorithm presented in this article is effective and is based a simple concept, even if its expression in algorithmic form is complex. Each character or byte in the input character or byte stream is directly encoded without change or modification in the compressed output stream. Then, if the given character or byte reappears or reoccurs in the next 2,040 characters or bytes, skipping characters or bytes that have already been encoded as reappearances or recurrences of previous bytes or characters, these reappearances or recurrences are encoded if the choice to encode the reappearances or recurrences saves space in the compressed output stream. Otherwise, no encoding is performed, and the next character or byte in the input character or byte stream is encoded without change or modification, and its reappearances or recurrences are possibly encoded if the choice to encode saves space, etc.

Consider the input character or byte stream 'ABAB'. The first character or byte in this input character or byte stream is 'A'. Another 'A' reappears

or reoccurs within the next 2,040 bytes or characters. In fact, it appears two characters later. This second occurrences of 'A' is not encoded because doing so will not save space. If we chose to encode the second appearance or occurrence of 'A', first we'd encode the first 'A' without change or modification, followed by the escape sequences, 00000000_2. The escape sequence indicates to the decompression algorithm that an encoding follows. If a natural 00000000_2 occurs in the input character or byte stream, 00000000_2 is encoded twice in the output stream to indicate that the 00000000_2 00000000_2 should not be interpreted as an encoding, but as a natural occurrence of 00000000_2. Next, following the escape sequence, a byte or character is used to specify the length of the encoding in bytes, which can be any number between 1 and 255. The encoding itself then follows the length of the encoding, which was preceded by the escape sequence.

In our example where our input character or byte stream is 'ABAB', and assuming we chose to encode the second appearance or occurrence of 'A' even though it doesn't save space, the encoding length would be 1 byte or character, and the encoding itself would be 010. But bits are represented in groups of 8 as bytes. So, appending 0's, the encoding would be 01000000. The left-most 0 in this encoding indicates that the character immediately following the first 'A' is not a reappearance or recurrences of 'A'. In fact, it's a 'B', not an 'A'. The 1 that then follows indicates that the character following 'B' is the first and only reappearance or recurrence of 'A' in the input character or byte stream.

Let's continue with this example, where our input character or byte stream is 'ABAB'. Assume we encoded the first and second appearance or occurrence of 'A', and we are now considering the first appearance or occurrence of 'B'. Assume also that even though it will not save space, we choose to encode all reappearances or recurrences of 'B' after the first 'B'. So, first 'B' would be encoded without modification or change in the output stream, followed by the escape sequence, 00000000_2, followed by the length of the stream, again 1, followed by the encoding. The encoding itself skips the 'A' that follows the first 'B' since this 'A' was encoded as a reappearance or recurrence after the first 'A'. So, our encoded sequence is simply 1. But bits are represented in groups of eight (8) as bytes, so appending 0's we get 10000000. The left-most one (1) encodes the second 'B' in our input byte or character stream.

A more complicated compression example is given in Appendix A.

When deciding whether or not to encode reappearances or recurrences of a character or byte in our input stream, we look at the following 2,040 characters or bytes in the input stream that have not already been encoded as reappearances or recurrences of a previous character or byte in the input stream. If we choose to encode, after the escape sequence, we specify the length of the encoding measured in bytes, a number between 1 and 255. The number zero (0) is reserved for the escape sequence, and if the number zero appears twice in-a-row, it specifies a natural 0 (zero) in our input stream. Notice that $255 * 8 = 2,040$. We look no father than 2,040 bytes or characters ahead, because an encoding of length 255, measures in bytes, represents 2,040 bits.

Figures 1 and 2 specify the compression algorithm. Figure 2 is the algorithm for COMPRESS, which uses the helper function ENCODE. Figure 1 is the algo-

rithm for ENCODE. Parameters passed to either function, either COMPRESS or ENCODE, which are preceded by **var** are passed by reference, so their values can change in the algorithms.

ENCODE (*uncompressed_data, i, claimed,* **var** *newly_claimed,* **var** *encoded,* **var** *length_stream)*

```
 1  byte_index ← 0
 2  bit_index ← 0
 3  j ← i + 1
 4  skipped ← 0
 5  encoded ← 0
 6  stream[0] ← 00000000₂
 7  while j − i ≤ 2040 and j + skipped < length(uncompressed_data) do
 8      if not claimed[j + skipped] then
 9          if uncompressed_data[j + skipped] = uncompressed_data[i] then
10              Set 2^bit_index bit of stream[byte_index] to binary 1
11              newly_claimed[j + skipped] ← true
12              encoded ← encoded + 1
13          else Set 2^bit_index bit of stream[byte_index] to binary 0
14          bit_index ← bit_index + 1
15          if bit_index > 7 then
16              bit_index ← 0
17              byte_index ← byte_index + 1
18              stream[byte_index] ← 00000000₂
19          j ← j + 1
20      else skipped ← skipped + 1
21  while byte_index ≥ 0 and stream[byte_index] = 00000000₂ do
22      byte_index ← byte_index - 1
23  length_stream ← byte_index + 1
24  return stream
```

Figure 1: ENCODE

2 Decompression algorithm

Let's consider decompressing the example given in the previous section. In the previous section we considered compressing the character or byte stream 'ABAB'. Assume we chose to encode the second and only reappearances or recurrences of both 'A' and 'B', even though it does not save space. Given this assumption, from the previous section, 'ABAB' is compressed as:

COMPRESS (*uncompressed_data*, **var** *length_compressed_data)*

```
1   length_compressed_data ← 0
2   for i ← 0 to length(uncompressed_data) – 1 do
3       claimed[i] ← false
4   for i ← 0 to length(uncompressed_data) – 1 do
5       if not claimed[i] then
6           claimed[i] ← true
7           compressed_data[length_compressed_data] ← uncompressed_data[i]
8           length_compressed_data ← length_compressed_data + 1
9           if uncompressed_data[i] = 00000000₂ then
10              compressed_data[length_compressed_data] ← 00000000₂
11              length_compressed_data ← length_compressed_data + 1
12          count ← 0
13          j ← i + 1
14          while count < 2040 and j < length(uncompressed_data) do
15              if not claimed[j] then
16                  count ← count + 1
17              newly_claimed[j] ← false
18              j ← j + 1
19          encoded ← 0
20          length_stream ← 0
21          stream ← ENCODE (uncompressed_data, i, claimed, newly_claimed, encoded,length_stream)
22          if (length_stream + 2) < encoded then
23              compressed_data[length_compressed_data] ← 00000000₂
24              length_compressed_data ← length_compressed_data + 1
25              compressed_data[length_compressed_data] ← length_stream
26              length_compressed_data ← length_compressed_data + 1
27              for j ← 0 to length_stream – 1 do
28                  compressed_data[length_compressed_data] ← stream[j]
29                  length_compressed_data ← length_compressed_data  + 1
30              count ← 0
31              j ← i + 1
32              while count < 2040 and j < length(uncompressed_data) do
33                  if not claimed[j] then
34                      count ← count + 1
35                  claimed[j] ← claimed[j] ∨ newly_claimed[j]
36                  j ← j + 1
37  return compressed_data
```

Figure 2: COMPRESS

```
A 00000000 00000001 01000000 B 00000000 00000001 10000000
```

This is our compressed input stream, and we will form a decompressed output stream. The 'A' is immediately read and placed into the decompressed output stream. Then 00000000_2 follows. This is the escape sequence, so we know an encoding follows, which encodes the reappearances or recurrences of 'A'. If another escape sequence, 00000000_2, followed the first escape sequence, it would really specify a natural 00000000_2 in the compressed input stream, and we would not assume that an encoding followed. Next is 00000001_2. This is the length of the encoding measured in bytes, namely 1 (one). Then the encoding itself follows, 01000000_2, an encoding of length one (1), measured in bytes. This encoding indicates that not the next character, but the character after the next charter is a reappearance or recurrence of 'A'. So this is our decompressed output stream so far: 'A?A?', where question marks (?'s) represent currently missing or unknown information.

Next we read 'B' from the compressed input stream. This is immediately placed in the decompressed output stream, so our decompressed output stream is currently 'ABA?'. Then the escape sequence, 00000000_2, follows, so we know an encoding follows, an encoding which encodes reappearances or recurrences of 'B'. Next is 00000001_2. This is the length of the encoding measured in bytes, namely 1 (one). Then the encoding itself follows, 10000000_2, an encoding of length one (1), measured in bytes. This encoding indicates the next question mark (?) in our decompressed output stream is a reappearance or recurrence of 'B'. So that completes our decompressed output stream: 'ABAB'. We have decompressed the compressed version of 'ABAB'.

A more complicated decompression example is given in Appendix B.

Figures 3 and 4 specify the decompression algorithm. Figure 4 is the algorithm for DECOMPRESS, which uses the helper function DECODE. Figure 3 is the algorithm for DECODE. Parameters passed to either function, either DECOMPRESS or DECODE, which are preceded by **var** are passed by reference, so their values can change in the algorithms.

3 Time complexity

Both the compression and decompression algorithms require time on the order of $O(n)$, where n specifies the linear size of the input. In other words, both algorithms are linear, and can compress and decompress data quickly.

3.1 Time complexity of the compression algorithm

As previously mentioned, the compression algorithm is linear with respect to the size of the input. COMPRESS requires the helper function ENCODE. Let's look at ENCODE first.

DECODE (*compressed_data*, **var** *i*, **var** *uncompressed_data*, **var** *j*, **var** *claimed*)

```
1  character ← compressed_data[i]
2  uncompressed_data[j] ← character
3  claimed[j] ← true
4  while j < length(uncompressed_data) and claimed[j] do
5      j ← j + 1
6  i ← i + 1
7  if compressed_data[i − 1] = 00000000₂ and compressed_data[i] = 00000000₂ then
8      i ← i + 1
9  if i < length(compressed_data) then
10     if compressed_data[i] = 00000000₂ and compressed_data[i + 1] ≠ 00000000₂ then
11         count ← 0
12         k ← j
13         while count < 2040 and k < length(uncompressed_data) do
14             if not claimed[k] then
15                 count ← count + 1
16             newly_claimed[k] ← false
17             k ← k + 1
18         i ← i + 1
19         encoding_length ← compressed_data[i]
20         j₂ ← j
21         for k = 0 to encoding_length − 1 do
22             i ← i + 1
23             encoded_byte ← compressed_data[i]
24             for l = 0 to 7 do
25                 if 2^l bit of encoded_byte set to binary 1 then
26                     uncompressed_data[j₂] ← character
27                     newly_claimed[j₂] ← true
28                 j₂ ← j₂ + 1
29                 while j₂ < length(uncompressed_data) and claimed[ j₂] do
30                     j₂ ← j₂ + 1
31         i ← i + 1
32         count ← 0
33         k ← j
34         while count < 2040 and k < length(uncompressed_data) do
35             if claimed[k] then
36                 count ← count + 1
37             claimed[k] ← claimed[k] ∨ newly_claimed[k]
38             k ← k + 1
```

Figure 3.6 DECODE

DECOMPRESS (*compressed_data, length_uncompressed_data)*

```
1  for i ← 0 to length_uncompressed_data – 1 do
2      claimed[i] ← false
3  i ← 0
4  j ← 0
5  while i < length(compressed_data) do
6      DECODE (compressed_data, i, uncompressed_data, j, claimed)
7      while j < length_uncompressed_data and claimed[j] do
8          j ← j + 1
9  return uncompressed_data
```

Figure 4: DECOMPRESS

ENCODE has two loop structures: the while loop at line 7, and the while loop at line 21. There are no other loop constructs in ENCODE. Both while loops do a constant amount of work, so the function ENCODE itself does a constant amount of work. Let's now examine COMPRESS.

In COMPRESS, the for loops at lines 2 and 4 do a linear amount of work, or $O(n)$ work. The while loop at line 14, the for loop at line 27, and the while loop at line 32, all do a constant amount of work. COMPRESS calls ENCODE on line 21, but remember ENCODE itself does a constant amount of work. Therefore, COMPRESS is linear or does a linear amount of work to compress input data. If n is the linear size of the input, COMPRESS requires time on the order of $O(n)$ to compress input data.

Figure 5 plots the time it takes to generate random data, and then compress it. The open circles in the plot represent real data to which the plotted line was fit. If the x-axis or the size of the input is n, and if the y-axis or time measured in seconds is t, then the equation of the line fitted to the real data is approximately:

$$t = 0.000077059n + 0.3867179$$

In other words, in the generation and compression of random data, the compression algorithm has overhead of approximately 0.3867179 seconds, and it takes approximately 0.00007705963 seconds to compress each byte in the random data.

3.2 Time complexity of the decompression algorithm

As previously mentioned, the decompression algorithm is linear with respect to the size of the input. DECOMPRESS requires the helper function DECODE. Let's look at DECODE first.

DECODE has six loop constructs: the while loop on line 4, the while loop on line 13, the for loop on line 21, the for loop on line 24, the while loop on line

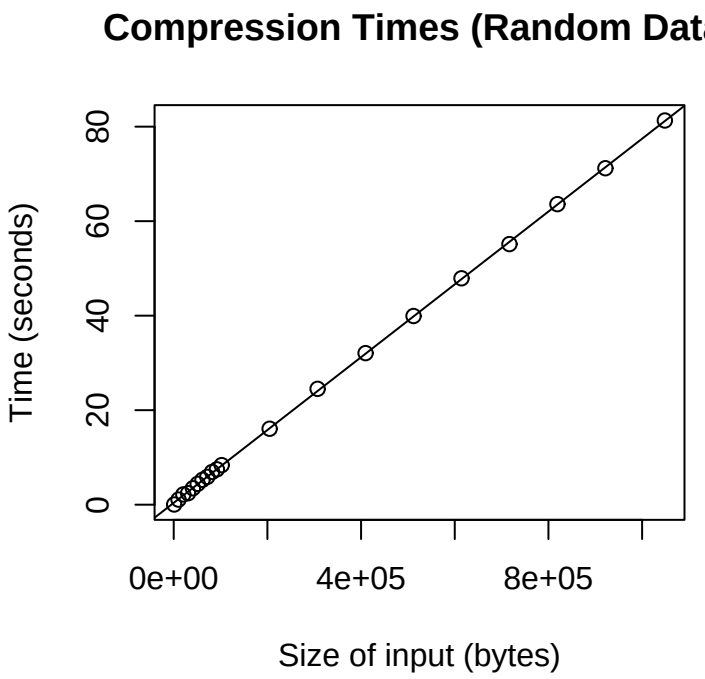

Figure 5: Size of input vs. Time (Random Data)

29, and the while loop on line 34. All six of these loop constructs do a constant amount of work. Therefore, the function DECODE itself does a constant amount of work. Let's now examine DECOMPRESS.

In DECOMPRESS, the for loop on line 1, and the while loop on line 5, both do a linear amount of work, or $O(n)$ work. The while loop on line 7 does a constant amount of work. DECOMPRESS calls DECODE on line 6, but remember DECODE itself does a constant amount of work. Therefore, DECOMPRESS is linear or does a linear amount of work to decompress input data. If n is the linear size of the input, DECOMPRESS requires time on the order of $O(n)$ to decompress input data.

Figure 6 plots the time it takes to generate random data, and then decompress it. The open circles in the plot represent real data to which the plotted line was fit. If the x-axis or the size of the input is n, and if the y-axis or time measured in seconds is t, then the equation of the line fitted to the real data is approximately:

$$t = 0.0000006301167n + 0.01284272$$

In other words, in the generation and decompression of random data, the decompression algorithm has overhead of approximately 0.01284272 seconds, and it takes approximately 0.0000006301167 seconds to decompress each byte in the random data.

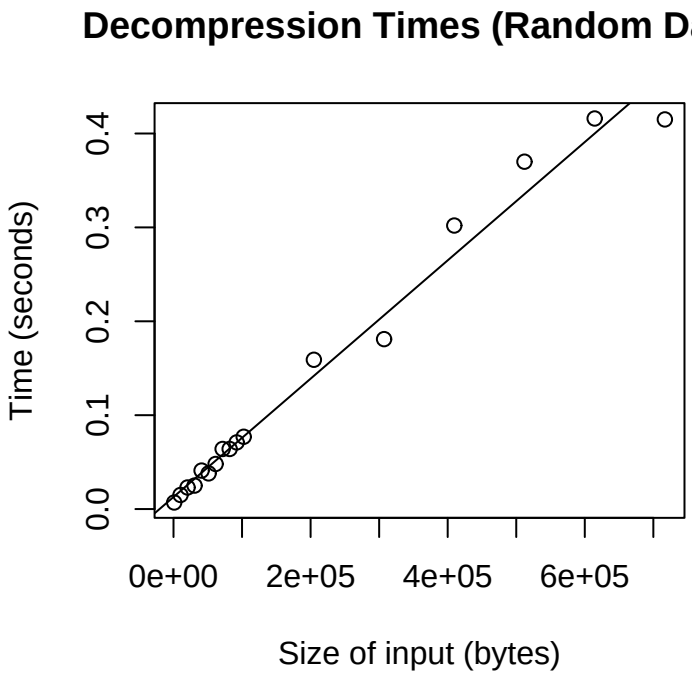

Figure 6: Size of input vs. Time (Random Data)

A Compression example

Consider the following character or byte stream, which we wish to compress.

```
Alan Alan Alan Alan Alan Alan Alan Alan Alan
```

There are 44 characters or bytes in this character or byte stream, and we wish to compress it, so it uses less memory. The first character in the character or byte stream is 'A'. This is immediately encoded into the compressed output stream. It is obvious, but this is our compressed output stream so far:

```
A
```

After this first 'A', additional A's occur 5 characters or bytes later, 10 characters or bytes later, 15 characters or bytes later, 20 characters or bytes later, 25 characters or bytes later, 30 characters or bytes later, 35 characters or bytes later, and 40 characters or bytes later. That's the other eight A's occurring in the character or byte stream. Each 'Alan' begins with a capital 'A', and there are eight Alan's following the first 'Alan'.

For each of the forty-three (43) characters following the first 'A', we can assign a binary digit. This binary digit is equal to zero (0_2) if the character is not an 'A', and is equal to one (1_2) if the character is equal to 'A'. Let's look at this stream of binary digits:

```
00001000 01000010 00010000 10000100 00100001 000
```

Next, there are three (3) zeros at the tail of this binary stream. But in any implementation, binary digits are represented as bytes, which are eight bits a piece. So, it our representation, this bit steam is actually:

```
00001000 01000010 00010000 10000100 00100001 00000000
```

So, this is the binary encoding of the forty-three (43) characters that follow the first 'A', represented as six (6) bytes. Next, we strip away trailing 00000000's from the bit or byte stream. These trailing 00000000's are unnecessary, and waste space should we choose to encode them later. So, our binary or byte stream is now:

```
00001000 01000010 00010000 10000100 00100001
```

It will take 5 characters or bytes of space to encode or represent these eight additional occurrences of 'A' following the first 'A'. But we also need to encode the escape sequence, 00000000_2, following the 'A' which we have already encoded, and after the escape sequence we need to specify the number of bytes or characters in the encoded byte stream, namely five (5). The number 5 in binary is 00000101_2. So, it takes $1 + 1 + 5 = 7$ characters to represent these eight additional A's. Since $7 < 8$, the algorithm chooses to encode in this format, to save space. So, here's our compressed output so far:

```
A 0000000 00000101 00001000 01000010 00010000 10000100
0010000100
```

The next unique character in our input is 'l'. It is immediately encoded:

```
A 0000000 00000101 00001000 01000010 00010000 10000100
00100001 l
```

Once again, there are eight more occurrences of 'l', but this time we skip all the additional A's previously encoded. Ignoring these already encoded A's, after the first 'l', l's occur 4 characters or bytes later, 8 characters or bytes later, 12 characters or bytes later, 16 characters or bytes later, 20 characters or bytes later, 24 characters or bytes later, 28 characters or bytes later, and 32 characters or bytes later. As before, here is how these recurrences are represented as a binary stream of digits:

```
00010001 00010001 00010001 00010001 00
```

There are forty-two (42) binary digits in this bit stream, but, again, in any implementation, bits are represented as bytes, so we append additional 0's to create five (5) bytes:

```
00010001 00010001 00010001 00010001 00000000
```

But again, the last byte, 00000000_2, encodes nothing, and would only waste space if we chose to encode it. Therefore, it is removed from the encoding. This leaves us with the following bit or byte stream:

```
00010001 00010001 00010001 00010001
```

Each of these binary numbers take one character or byte to represent in compressed format. But once again, after the 'l' we need to include the escape sequence, 00000000_2, and the number of bytes in the bit or byte stream, namely four (4) or 00000100_2. So a total of 6 bytes is required to represent these additional eight occurrences of 'l'. Since $6 < 8$, the encoded representation is chosen. So, here's our compressed output:

```
A 00000000 00000101 00001000 01000010 00010000 10000100
00100001 l 00000000 00000100 00010001 00010001 00010001
00010001
```

The next unique character in our input is 'a'. It is immediately encoded:

```
A 00000000 00000101 00001000 01000010 00010000 10000100
00100001 l 00000000 00000100 00010001 00010001 00010001
00010001 a
```

Now eight additional a's occur in the input. We ignore already encoded characters, namely, the last eight A's and the last eight l's. Ignoring these already encoded letters, after the first 'a', additional a's occur 3 characters or bytes later, 6 characters or bytes later, 9 characters or bytes later, 12 characters or bytes later, 15 characters or bytes later, 18 characters or bytes later, 21 characters or bytes later, and 24 characters or bytes later. Again, these recurrences are represented as a binary stream of digits:

```
00100100 10010010 01001001 0
```

But once again, in any implementation, bits are represented as bytes, so the tailing zero (0) is really represented by 8-bits, with zero's (0's) appended, or:

```
00100100 10010010 01001001 00000000
```

But once again, the tailing byte, 00000000_2, encodes nothing, and would waste space if we chose to encode it, so it is trimmed from the binary bit or byte stream, leaving:

```
00100100 10010010 01001001
```

Once again, these three bytes or characters must be precede by the escape sequence, 00000000_2, and the number of bytes in the stream, namely three (3) or 00000011_2. Therefore, $1 + 1 + 3 = 5$ characters are necessary to represent the eight additional l's. Since $5 < 8$, we choose to encode the eight additional a's in this manner, so now the compress output looks like:

```
A 00000000 00000101 00001000 01000010 00010000 10000100
00100001 l 00000000 00000100 00010001 00010001 00010001
00010001 a 00000000 00000011 00100100 10010010 01001001
```

The next unique character in our input is 'n'. It is immediately represented in our output:

```
A 00000000 00000101 00001000 01000010 00010000 10000100
00100001 l 00000000 00000100 00010001 00010001 00010001
00010001 a 00000000 00000011 00100100 10010010 01001001
n
```

After this first occurrence of 'n', there are eight additional occurrences of 'n'. Ignoring the characters that have already been encoded, 'A', 'l', and 'a', after the first 'n', n's occur after 2 characters or bytes, 4 characters or bytes, 6 characters or bytes, 8 characters or bytes, 10 characters or bytes, 12 characters or bytes, 14 characters or bytes, and 16 characters or bytes. These recurrences are represented as:

```
01010101 01010101
```

These two binary numbers are each represented by a single character or byte, but we must also include the escape sequence, 00000000_2, and the number of characters or bytes in the stream, namely two (2), or 00000010_2. Therefore, $1 + 1 + 2 = 4$ characters or bytes are needed to represent these additional eight occurrences of 'n'. Since, $4 < 8$ we choose to encode. Our output is now:

```
A 00000000 00000101 00001000 01000010 00010000 10000100
00100001 l 00000000 00000100 00010001 00010001 00010001
00010001 a 00000000 00000011 00100100 10010010 01001001
n 00000000 00000010 01010101 01010101
```

The next unique character in our input is the space character, ' '. It is immediately represented in our output:

```
A 00000000 00000101 00001000 01000010 00010000 10000100
00100001 l 00000000 00000100 00010001 00010001 00010001
00010001 a 00000000 00000011 00100100 10010010 01001001
n 00000000 00000010 01010101 01010101 ' '
```

After this first space, ' ', seven additional spaces occur. Ignoring characters that have already been encoded, 'A', 'l', 'a', and 'n', after the first space, ' ', additional spaces occur after 1 character or byte, 2 characters or bytes, 3 characters or bytes, 4 characters or bytes, 5 characters or bytes, 6 characters or bytes, and 7 characters or bytes. This can be represented as:

```
1111111
```

But in any implementation, bits are represented in groups of 8 as bytes, so we need to add a single zero (0) to the end of this stream to make it a full byte:

```
11111110
```

This binary number will be represented as a single character or byte in the output. As usual, we'll also have to include the escape sequence, 00000000_2, after the first space, and the number of bytes in the binary bit or byte stream, namely one (1), or 00000001_2. So, it'll take $1 + 1 + 1 = 3$ characters or bytes to represent these additional seven spaces, ' '. Since $3 < 7$, we encode in this fashion. Here's our output:

```
A 00000000 00000101 00001000 01000010 00010000 10000100
00100001 l 00000000 00000100 00010001 00010001 00010001
00010001 a 00000000 00000011 00100100 10010010 01001001
n 00000000 00000010 01010101 01010101 ' ' 00000000 00000001
11111110
```

Now the entire input character or byte stream has been compressed or encoded. Remember, the input character or byte stream was 44 characters or bytes in length. In comparison, the compressed output is only 30 characters of bytes in length. Compression has been successful.

B Decompression example

Let's take the compressed output from the example in Appendix A, and decompress it. Remember, the compressed output from this example looked like:

```
A 00000000 00000101 00001000 01000010 00010000 10000100
00100001 l 00000000 00000100 00010001 00010001 00010001
00010001 a 00000000 00000011 00100100 10010010 01001001
n 00000000 00000010 01010101 01010101 ' ' 00000000 00000001
11111110
```

The non-encoded 'A' is read and placed in our decompressed output. Here's our decompressed output so far. Periods, '.', represent empty or unassigned spaces or characters. Note that we assume we know the size of the decompressed output before we decompress the input.

```
A.........................................
```

Next, the escape sequences follows, 00000000_2, and then the length of the encoded sequence measured in bytes is given, 00000101_2, or five (5). The five bytes that then follow specify additional occurrences of 'A' in our decompressed output. We decode these, giving:

```
A...A...A...A...A...A...A...A...A...
```

Next, an 'l' occurs, and this is immediately added to the decompressed output:

```
Al...A....A....A....A....A....A....A....A...
```

After the 'l', the escape sequence occurs, 00000000_2, followed by the number of bytes in the encoded sequence, 00000100_2, or four (4). The four bytes that then follow specify additional occurrences of 'l' in our decompressed output. We decode these, giving:

```
Al...Al...Al...Al...Al...Al...Al...Al...Al..
```

Next, an 'a' occurs, and this is immediately added to our decompressed output:

```
Ala..Al...Al...Al...Al...Al...Al...Al...Al..
```

The escape sequence, 00000000_2, then follows the 'a', and after the escape sequence, the number of bytes in the encoding is given, 00000011_2, or three (3). The three bytes that then follow specify additional occurrences of 'a' in the decompressed output. We decode these, giving:

```
Ala..Ala..Ala..Ala..Ala..Ala..Ala..Ala..Ala.
```

Next, an 'n' occurs, and this is immediately added to our decompressed output:

```
Alan.Ala..Ala..Ala..Ala..Ala..Ala..Ala..Ala.
```

The escape sequence, 00000000_2, follows the 'n', and after the escape sequence, the number of bytes in the encoding is given, 00000010_2, or two (2). The two bytes that then follow specify additional occurrences of 'n' in the decompressed output. We decode these, giving:

```
Alan.Alan.Alan.Alan.Alan.Alan.Alan.Alan.Alan
```

Next, a space, or ' ', occurs, and this is immediately added to our decompressed output:

```
Alan Alan.Alan.Alan.Alan.Alan.Alan.Alan.Alan
```

The escape sequence, 00000000_2, follows the space, and after the escape sequence, the number of bytes in the encoding is given, 00000001_2, or one (1). The one byte that then follows specifies additional occurrences of the space, or ' ', in the decompressed output. We decode these, giving:

```
Alan Alan Alan Alan Alan Alan Alan Alan Alan
```

We have decompressed the compressed output from Appendix A.