

DATA COMPRESSION ALGORITHM DESIGN

Author: Isaac Shiells Thomas

Email: isaacshiellsthomas.work@proton.me

Contents

1	Context	1
1.1	Assumptions	1
2	Algorithm	1
2.1	Case 1: 3 or More Sequential Bytes Found	2
2.2	Case 2: 2 or Less Sequential Bytes Found	3

1 Context

This is an algorithm designed to compress a given buffer of data in the form of a byte array.

The algorithm lives within a function. This function is called with two arguments; a pointer to a data buffer in the form of a byte array (`data_ptr`), and the number of bytes to compress in the form of an integer (`data_size`). After the function executes, the data in the buffer will be modified and the size of the modified buffer will be returned.

It is not expected for the algorithm to reallocate memory for the new size of the buffer. The space after the requested number of bytes to be compressed will not be altered, and the trailing space — the space saved by the compression — will be zeroed.

1.1 Assumptions

1. The `data_ptr` will point to an array of bytes. Each byte will contain a number from 0 to 127 (0x00 to 0x7F). Additionally, it is common for the data in the buffer to have the same value repeated multiple times in a row.
2. The compressed data needs to be decompressable. A accompanying function that will decompress the data must exist.

2 Algorithm

This algorithm will compare chunks of 7 *semi-unique* bytes at a time, starting from the 0th byte of the byte array. First, the initial byte is recorded, the algorithm then scans the each subsequent byte until a byte of a different value is found. At that point, the quantity of the first byte, and it's value, are recorded as the *first* of the *7-byte-chunk*, and then the algorithm continues for the next byte (the one that ended the sequential streak of the first byte). The process is repeated until 7 bytes are found. If the algorithm comes across a byte that is already present as a previous value in the *7-byte-chunk*, then it treats it *no differently* than if it was completely new. The algorithm only cares about separating and condensing the sequential streams of bytes (bytes that all have the same value and appear in a row). It does not care if the *7-byte-chunk* consists of only two different bytes, alternating back and forth.

Now that the *7-byte-chunk* has been filled, the algorithm will handle two cases differently: 1) When their exists at least one byte that appears three or more times in a row, and 2) when there doesn't.

In both cases, a header byte will be used to facilitate the decoding process. [Case 1](#) will use a full header byte while [case 2](#) will use a partial header byte (details explained below). Despite their differences, the two cases use a single high order bit to declare to the decoder which type of compression was used for the following block of bytes. A high order bit of value “0” indicates that the sub-algorithm defined in case 1 is used, while a high order bit of value “1” indicates that the sub-algorithm defined in case 2 is used.

The two cased are defined as follows:

2.1 Case 1: 3 or More Sequential Bytes Found

This case occurs when at least one of the bytes in the *7-byte-chunk* occurred **three times or more sequentially**. Bytes of equal value within the *7-byte-chunk* but are immediately preceded and followed by bytes of differing values, as well as like bytes that appear in pairs **do not** qualify for this case.

For example, consider the following streams:

$$(0x04, 0x11, 0x11, 0x11, \dots), \quad (1)$$

$$(0x04, 0x11, 0x04, 0x51, \dots), \quad (2)$$

and

$$(0x6C, 0x39, 0x6C, 0x6C, 0x44, \dots). \quad (3)$$

The first stream (Eq. 1) would include the byte **0x11** with a quantity of **3** as the second byte type of its *7-byte-chunk*. This stream would qualify for this case.

The second stream (Eq. 2) would include the byte **0x04** with a quantity of **1** as the first byte type of its *7-byte-chunk*, and would include the (same) byte **0x04** with a quantity of **1** as the third byte type of its *7-byte-chunk*. This stream would **not** qualify for this conditional case (see [case 2](#) where it would qualify).

The third stream (Eq. 3) would include the byte **0x6C** with a quantity of **1** as the first byte type of its *7-byte-chunk*, the byte **0x39** with a quantity of **1** as the second byte type, and the byte **0x6C** with a quantity of **2** as the third byte type of its *7-byte-chunk*. Note how the bytes **0x6C** are **not** aggregated into one. This stream would **not** qualify for this conditional case (see [case 2](#) where it would qualify).

This case uses a header byte to provide information about the following set of bytes. This header byte is a full header byte, meaning that all of the bits of the header are used to convey information other than the data. This header byte has the following encoding:



The *Compression Type Bit* indicates that the following set of bytes is encoded using the sub-algorithm of this case. Each bit of the *Repeated Byte Indicators* encode whether the byte at that positional index is a sequential encoded byte or not.

If the bit at position 1 (on Eq. 4) is set to “0”, then the byte is an individual byte and does not have any multiplicity. Otherwise, if the bit is set to “1”, then this indicates that the byte *does* have multiplicity and the pair of bytes at that relative index should be considered: The first byte of the pair is the actual value of the byte. The second byte of the pair is the number of times this byte appears sequentially in the original data.

For example, the stream

$$(0x03, 0x74, 0x04, 0x04, 0x35, 0x35, 0x64, 0x64, 0x64, 0x64, 0x00, 0x00, 0x00, 0x00, 0x00, 0x56) \quad (5)$$

would become the *7-byte-chunk* represented by the array of 2-tuples where the first value of each 2-tuple represents the byte value, and the second value represents the multiplicity of that byte (the number of times it occurred in a row). The position of the 2-tuple in the array represents position of that byte relative to the other 2-tuples in the *7-byte-chunk*. As follows:

$$((0x03, 1), (0x74, 1), (0x04, 2), (0x35, 2), (0x64, 4), (0x00, 5), (0x56, 1),) \quad (6)$$

when encoded back into a byte array, using this algorithm, we would get:

$$(0b00011110, 0x03, 0x74, 0x04, 0x00, 0x35, 0x00, 0x64, 0x02, 0x00, 0x03, 0x56). \quad (7)$$

This resulting compressed data is 12 bytes in length, whereas the original stream of data was 16 bytes in length. That's a 25% decrease, and thus a successful compression.

You may notice that in the compressed stream of bytes (Eq. 7) that the multiplicity bytes do not match the values found in the corresponding 7-byte-chunk (Eq. 6). In fact, they are all a value of 2 lower than what might have been expected. This is because in the header byte, we already indicate when a pair of bytes will be present, and this only occurs when the byte value has a multiplicity of at least 2. Thus, we can set the multiplicity byte to start counting at 2 (represented by the value 0) so that we can store a byte with a multiplicity of up to 257 instead of only up to a multiplicity of 255; slightly increasing our potential for compression.

2.2 Case 2: 2 or Less Sequential Bytes Found

In the