Data Compression Algorithm Design

Author: *Isaac Shiells Thomas*
Email: isaacshiellsthomas.work@proton.me

Contents

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.

1.2 Goals

The first goal of this algorithm will be to compress the data by utilizing the fact that it can be commonly expected that bytes will appear multiple times sequentially. The main tactic will be to condense repeated sequential bytes into an encoding of two bytes where the first byte is the original value and the second byte is the number of times in a row that byte was repeated.

The second goal will be that, even in the worse case scenario, this algorithm will **not** increase the length of the data and at worst will result in a compressed data length equal to the inputted data length (`data_size`) when measured in bytes.

2 Terms and Definitions

Sequential Byte A *sequential byte* is a set of bytes in a stream that appears twice or more in a row with the exact same value. An instance of a byte that appears twice or more in a row is called *sequential*.

Byte Multiplicity When an instance of a byte is sequential, its *multiplicity* is the *number of times* that byte appears in a row. In the byte stream (`0x64`, `0x4A`, `0x64`, `0x64`, `0x64`), the second instance of the byte `0x64` has a *multiplicity* of **3**. A byte that is *not sequential* has a *multiplicity* of **1**.

Semi-Unique Byte Set A set of bytes where order matters and each byte may appear more than once, but not more than once in a row.

3 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 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, semi-unique, 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.

If the input data buffer does not have enough bytes to fill the *7-byte-chunk*, then any remaining slots are filled with entries containing a data value of "`0xFF`" and a multiplicity value of 0. This combo of impossible values depicts that the end of the data has been reached.

Now that the *7-byte-chunk* has been filled, the algorithm will handle two cases differently:

1) When their exists at least one sequential byte that appears with a multiplicity of 3 or more, **or** the 7-byte-chunk has trailing values with multiplicity 0.
2) Otherwise.

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:

3.1 Case 1: Sub-Algorithm 1

This case occurs when at least one of the bytes in the *7-byte-chunk* occurred **three times or more** *sequentially* **<u>or</u>** the number of bytes remaining in the input buffer is not enough to fill the *7-byte-chunk* (i.e. EOF).

For example, consider the following streams:

$$(\texttt{0x04}, \texttt{0x11}, \texttt{0x11}, \texttt{0x11}, \ldots), \tag{1}$$

$$(\texttt{0x04}, \texttt{0x11}, \texttt{0x04}, \texttt{0x51}, \ldots), \tag{2}$$

$$(\texttt{0x04}, \texttt{0x11}, \texttt{0x04}, \texttt{0x51}), \tag{3}$$

and

$$(\texttt{0x6C}, \texttt{0x39}, \texttt{0x6C}, \texttt{0x6C}, \texttt{0x44}, \ldots). \tag{4}$$

The first stream (Eq. 1) would include the byte `0x11` with a multiplicity of **3** as the second byte type of its 7-byte-chunk. This stream would qualify to use the sub-algorithm of this case.

The second stream (Eq. 2) would include the byte `0x04` with a multiplicity of **1** as the first byte type of its 7-byte-chunk, and would include the (same) byte `0x04` with a multiplicity of **1** as the third byte type of its 7-byte-chunk. This stream would **not** qualify (without knowing the hidden bytes) for this conditional case (see case 2 where it would qualify).

The third stream (Eq. 3) (which is the same as the second stream, but with more information) would not have enough bytes to fill the *7-byte-chunk* and would thus qualify for this case.

The fourth stream (Eq. 4) 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).

3.1.1 Header Byte

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 the bits of the header are used to convey information other than the data. This header byte has the following encoding:

$$\underset{\text{Compression Type Bit}}{\underline{0}}\overbrace{\underset{0}{\_}\,\underset{1}{\_}\,\underset{2}{\_}\,\underset{3}{\_}\,\underset{4}{\_}\,\underset{5}{\_}\,\underset{6}{\_}}^{\substack{\text{Repeated Byte}\\ \text{Indicators}}} \tag{5}$$

The ***Compression Type Bit*** indicates that the following set of bytes is encoded using the case-1 sub-algorithm.

Each bit of the ***Repeated Byte Indicators*** encode whether the byte at that positional index is a sequential byte or not.

If the bit at position 0 (on Eq. 5) is set to “`0`”, then the byte is an individual byte and does not have any multiplicity ($M_{\text{ultiplicity}} \leq 1$). Otherwise, if the bit is set to “`1`”, then this indicates that the byte *does* have multiplicity ($M_{\text{ultiplicity}} \geq 2$) and the pair of bytes at that relative index should be considered: The first byte of the pair is the data value of the byte. The second byte of the pair is the number of times this byte appears sequentially in the original data. This repeats for all the other indices of *repeated byte indicators*. This is repeated for every index of the *7-byte-chunk*.

3.1.2 Encoding

Data bytes are encoded depending on their multiplicity as well as if there are more bytes to follow.

For each byte instance from the *7-byte-chunk*, if the byte has a multiplicity of 2 or greater, then the bit at its index inside the header byte (Eq. 5) is set to “`1`” and is encoded as a pair of bytes where the first byte is the data value and the second byte is its multiplicity minus 2 (since this is the minimum value for this pair format). If the byte instance instead has a multiplicity exactly 1, then the bit at its index inside the header byte (Eq. 5) is set to “`0`” and is encoded as a solo byte representing just the data value. If the byte instance has a multiplicity of 0 (i.e. it does not exist and indicates the end of the source data buffer) then it is excluded and not encoded into the output buffer.

Since the input data values only range from `0x00` to `0x7F`, this means that the top high order bit is always set to “`0`” for every encoded data byte. Since this bit can is not important to the actual data value, we instead will use it to indicate whether there is another data byte to follow (whether it’s a pair or solo), or if the end of the buffer has been reached. If there is a data byte to follow the current data byte, then this top bit is set to “`1`”, if there is not a byte to follow, then this top bit is set to “`0`”. If the current data byte (pair or solo) is the 7th and final data byte, then this top high order bit is ignored.

Note: There will never be a case where a 7-byte-chunk consists of all void bytes, thus we do not need to worry about the first byte not existing (i.e. each byte set will always include at least one data byte).

3.1.3 Encoding Example

As an example, the stream

$$(\texttt{0x03}, \texttt{0x74}, \texttt{0x04}, \texttt{0x04}, \texttt{0x35}, \texttt{0x35}, \texttt{0x64}, \texttt{0x64}, \texttt{0x64}, \texttt{0x64}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x56}) \quad (6)$$

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:

$$((\texttt{0x03}, 1), (\texttt{0x74}, 1), (\texttt{0x04}, 2), (\texttt{0x35}, 2), (\texttt{0x64}, 4), (\texttt{0x00}, 5), (\texttt{0x56}, 1)). \quad (7)$$

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

$$(\texttt{0b00011110}, \texttt{0x03}, \texttt{0x74}, \texttt{0x04}, \texttt{0x00}, \texttt{0x35}, \texttt{0x00}, \texttt{0x64}, \texttt{0x02}, \texttt{0x00}, \texttt{0x03}, \texttt{0x56}). \quad (8)$$

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. 8) that the *multiplicity bytes* do not match the values found in the corresponding 7-byte-chunk (Eq. 7). 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.

3.1.4 Worst Case Scenario

For this case of the algorithm to run, the *7-byte-chunk* must at least have one sequential byte with a multiplicity of 3. In the worst case scenario, there is one singular byte with a multiplicity of 3, and all other bytes have a multiplicity of 1 or 2.

The first byte outputted is the header info byte, this increases the output buffer length by 1.

For all the bytes of multiplicity 1, they are not encoded, and their byte is simply re-added to the stream at their appropriate index. Thus, for each of these bytes, the length of the output remains the same.

For all the bytes of multiplicity 2, they first output the byte of their own value, then followed by the byte encoding the quantity of bytes condensed. Since two bytes were encoded back into two bytes, the length of the output remains the same.

For the singular byte of multiplicity 3, this byte is encoded just the same as was done for bytes of multiplicity 2, except the quantity byte has a value of 3. Thus, we encoded 3 bytes from the input down to 2 bytes in the output, resulting in an output buffer length decrease of 1.

The decrease of 1 byte from the singular multiplicity 3 byte counteracts the increase of 1 byte from the header info byte, resulting in a final output buffer length equal to that of the input buffer length.

Thus, even in the worse case scenario, this component of the algorithm does not produce a compressed data buffer with a length greater than the uncompressed data buffer.

3.2 Case 2: 2 or Less Sequential Bytes Found

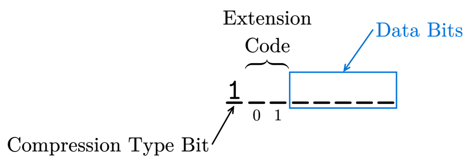
This case occurs when all the bytes in the *7-byte-chunk* are of multiplicity 1 or 2.

Since there are no sequential bytes from the 7-byte-chunk that can be compressed by encoding their multiplicity, this sub-algorithm uses the fact that every data byte of the source data has values from `0x00` to only `0x7F` to *condense* the data by removing the unused high-order-bit from every data byte. This sub-algorithm becomes more efficient when it can condense more bytes at once so it can aggregate all the "saved" high-order-bits together into the saving of an entire byte. To do this, after meeting the condition for using this sub-algorithm from the contents of the *7-byte-chunk*, it will look for additional bytes *past* those in the 7-byte-chunk to determine if there are more bytes with a multiplicity of less than three that can be encoded into this byte set with the desired result of saving an entire byte. Since there are only certain multiples of bytes that result in saving whole bytes, the number of bytes additionally collected into this encoding are specifically chosen and are outlined below.

3.2.1 Header Byte

Like case 1, this case also uses a header byte to provide information about the following set of bytes. This header byte is a *partial* header byte, meaning that a portion of the bits are used to encode information, and the rest are used to store data.

This header byte is encoded as follows:



(9)

The ***Compression Type Bit*** indicates that the following set of bytes is encoded using the *case-2 sub-algorithm*.

The ***Extension Code*** is the encoding that indicates how many *extra* bytes will be included in this byte set. Since only certain quantities of input bytes accumulate to result in saving a whole byte, these encodings map to specific values (see below).

The ***Data Bits*** are the initial bits of the compressed bytes from the original data. That is, the high order bits of the first byte of the 7-byte-chunk, excluding the first bit that is uniformly "0" among all the source data bytes.

3.2.1.1 Extension Codes

Extension Code		# Input Bytes	# Output Bytes	Trailing Spare Bits	Explanation
Bit 0	Bit 1				
0	0	7	7	4	This is the default encoding. This is the encoding that insures that this algorithm does not increase the number of bytes in the resulting output buffer compared to the length of the input buffer.
0	1	11	10	0	This is the number of input bytes required to save **1** whole byte in the output.
1	0	18	16	0	This is the number of input bytes required to save **2** whole bytes in the output.
1	1	25	22	0	This is the number of input bytes required to save **3** whole bytes in the output.

3.2.2 Encoding Example

3.2.2.1 Example 1

Consider the input byte array of

$$\begin{gathered}(\\ \texttt{0x03}, \texttt{0x74}, \texttt{0x04}, \texttt{0x1A}, \\ \texttt{0x1A}, \texttt{0x35}, \texttt{0x64}, \texttt{0x00}, \\ ).\end{gathered} \tag{10}$$

From this, the first *7-byte-chunk* of 2-tuples (*value*, *multiplicity*) would be constructed as:

$$((\texttt{0x03}, 1), (\texttt{0x74}, 1), (\texttt{0x04}, 1), (\texttt{0x1A}, 2), (\texttt{0x35}, 1), (\texttt{0x64}, 1), (\texttt{0x00}, 5), (\texttt{0x56}, 1)). \tag{11}$$

Since no byte instance of the *7-byte-chunk* (Eq. 11) has a multiplicity of 3 or greater, this byte set qualifies for this case.

3.2.2.2 Example 2

$$\begin{aligned}
&(\\
&\quad \texttt{0x03}, \texttt{0x74}, \texttt{0x04}, \texttt{0x1A}, \\
&\quad \texttt{0x1A}, \texttt{0x35}, \texttt{0x64}, \texttt{0x00}, \\
&\quad \texttt{0x35}, \texttt{0x35}, \texttt{0x56}, \texttt{0x0C}, \\
&\quad \texttt{0x1B}, \texttt{0x1B}, \texttt{0x1B}, \texttt{0x0C}, \\
&\quad \texttt{0x1B} \\
&).
\end{aligned} \tag{12}$$