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. An 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 pairs 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 evaluate sequential bytes one at a time and encode them.

First, for all the bytes in the input buffer, the algorithm will group each byte into a pair of values: The value of the byte itself, and the number of times that byte appeared in a row — its *multiplicity*. If the same byte appears multiple times but is separated by one or more bytes of a different value, then those same-bytes are treated as different *instances* of that byte.

For example, consider the following input data buffer:

$$(\texttt{0x03}, \texttt{0x74}, \texttt{0x1A}, \texttt{0x1A}, \texttt{0x64}, \texttt{0x64}, \texttt{0x64}, \texttt{0x64}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x1A}). \tag{1}$$

This would become the array of 2-tuples:

$$((\texttt{0x03}, 1), (\texttt{0x74}, 1), (\texttt{0x1A}, 2), (\texttt{0x64}, 4), (\texttt{0x00}, 5), (\texttt{0x1A}, 1)). \tag{2}$$

For each 2-tuple, the first value is the value of the data byte, and the second value is its multiplicity.

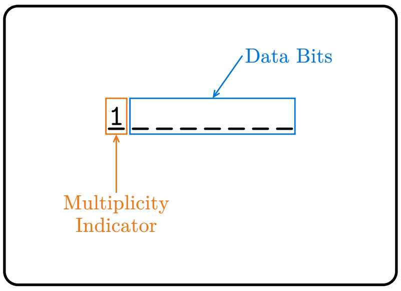

Figure 1: Encoded Data Byte

From here, the algorithm utilized the fact that the values for each data byte only range from `0x00` to `0x7F` and all have the same value for the top high-order-bit of their byte: "`0`".

For each byte-instance in the array of 2-tuples, the encoding differs depending on whether the byte-instance has a multiplicity equal to 1, or a multiplicity equal to 2 or greater. In both cases, the 7 low-order bits of the data-byte are encoded into the matching 7 low-order bits of the encoded-data-byte. Now, if the multiplicity of the byte instance is equal to 1, then the *multiplicity indicator* bit is set to "`0`" and the encoding is complete. Alternatively, if the multiplicity of the byte-instance is equal to 2 or greater, then the *multiplicity indicator* bit is set to "`1`" and the encoded-byte is written. In the *following* byte, the value of the multiplicity for this byte-instance is written, where a value of `0x00` encodes a multiplicity of 2, and `0xFF` encodes a multiplicity of 257. Writing this value-multiplicity byte pair completes the encoding for this second case.

Since a value-multiplicity pair is only written when a given byte-instance has a multiplicity greater than 1, the addition of the extra byte to encode the multiplicity will never replace what was a single byte from the original data with the pair of bytes, it will only ever replace sequences of the same byte repeated twice or more with the pair. This insures that in the worst case scenario — where no byte in the input data is ever found more than twice in a row — the resulting compressed data buffer will not be of greater length than the source data buffer. Thus, this satisfies our second goal.

4 Example

Consider the following inputted array of bytes:

$$\begin{pmatrix} \texttt{0x03}, \texttt{0x74}, \texttt{0x04}, \texttt{0x04}, & \texttt{0x04}, \texttt{0x35}, \texttt{0x35}, \texttt{0x64}, & \texttt{0x64}, \texttt{0x64}, \texttt{0x64}, \texttt{0x00}, \\ \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, \texttt{0x00}, & \texttt{0x56}, \texttt{0x45}, \texttt{0x56}, \texttt{0x56}, & \texttt{0x56}, \texttt{0x09}, \texttt{0x09}, \texttt{0x09} \end{pmatrix}. \tag{3}$$

This would become the array of value-multiplicity 2-tuples:

$$\begin{pmatrix} (\texttt{0x03}, 1), (\texttt{0x74}, 1), (\texttt{0x04}, 3), (\texttt{0x35}, 2), (\texttt{0x64}, 4), \\ (\texttt{0x00}, 5), (\texttt{0x56}, 1), (\texttt{0x45}, 1), (\texttt{0x56}, 3), (\texttt{0x09}, 3) \end{pmatrix}. \tag{4}$$

By performing the compression algorithm, we get:

$$\begin{pmatrix} \texttt{0x03}, \texttt{0x03}, \texttt{0x74}, \texttt{0x74}, \texttt{0x04}, \texttt{0x04}, \texttt{0x35}, \texttt{0x35}, \texttt{0x64}, \texttt{0x64}, \\ \texttt{0x00}, \texttt{0x00}, \texttt{0x56}, \texttt{0x56}, \texttt{0x45}, \texttt{0x45}, \texttt{0x56}, \texttt{0x56}, \texttt{0x09}, \texttt{0x09} \end{pmatrix}. \tag{5}$$

It can be pretty hard to read byte-by-byte and try to discern the changes. To make it easier, here is a diagram that represents the process followed by the algorithm for this example:

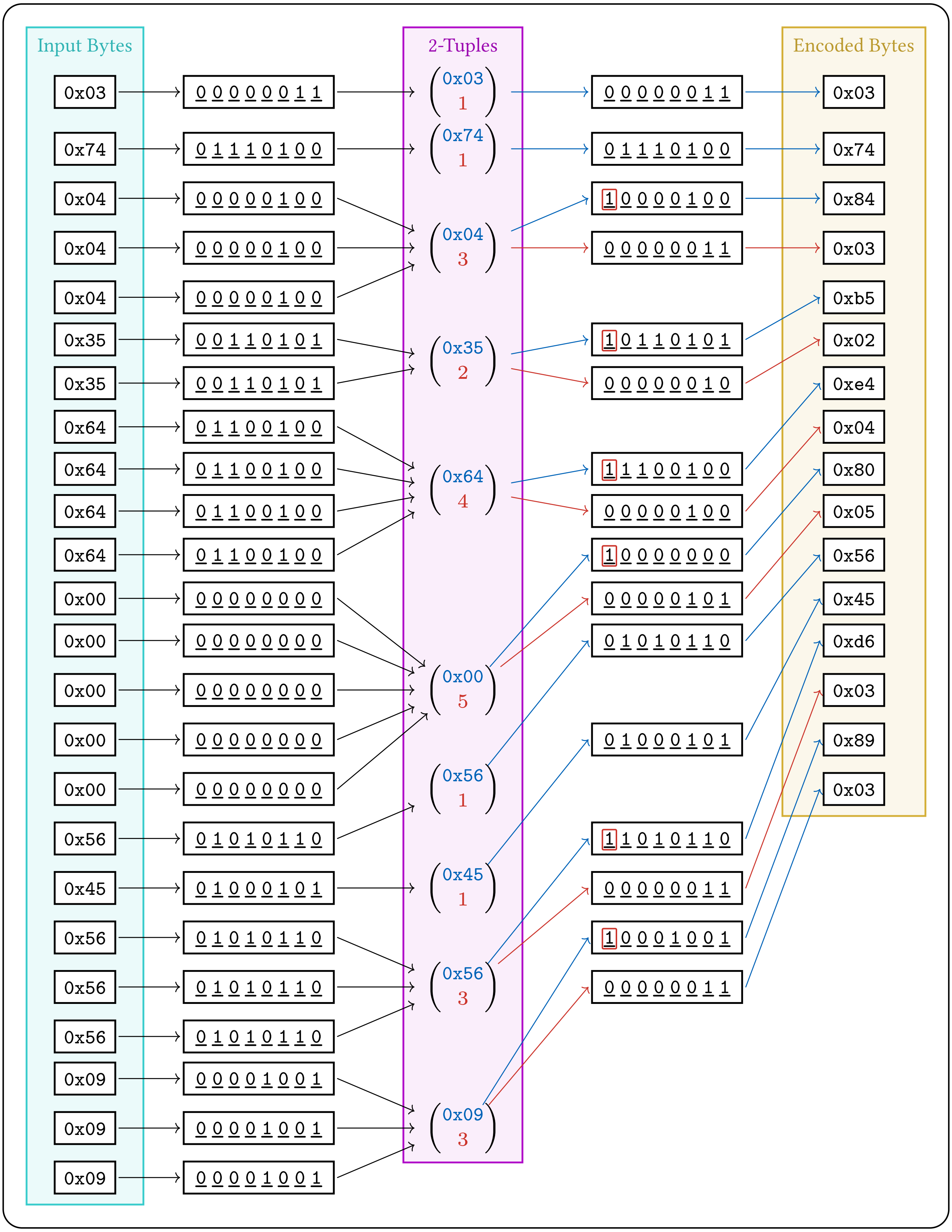

Figure 2: Algorithm Example