Programming Assignment

The Hyve

April 30, 2018

1 General Instructions

This document describes a programming assignment with a simple algorithmic and I/O component.

You should have received a preparatory text a week or so before starting this assignment, with some general hints on doing binary input/output and reading and writing data to and from standard input and output. This assignment will assume you are familiar with these terms and you know how to use these features.

The assignment is designed to be completed in 3 hours, though an extension of 1 hour can be granted on request from the applicant. Writing the algorithm component should be possible in well under this time. However, it is expected that the code be well-structured, minimally documented and unit tested. Time allowing, integration or functional tests would be regarded favorably. The objective here is not to create an "enterprise" *Hello World* program full of design patterns [1], yet a straightforward, purely procedural implementation with no abstractions would fall short of the target of this assignment.

Any programming language is acceptable, provided that the program uses the object oriented paradigm, in particular polymorphism.

The program can be distributed in any way the applicant wants. It is suggested that the code be pushed to a public git server (e.g. one on Github or Bitbucket), but this is not a requirement. Binaries are not necessary; in case a scripting language is not used, the applicant should include simple instructions on how to compile the supplied program.

2 Description of the Problem

In this assignment, you will be working on a very simple data compression scheme. Not everything will be fully specified. The applicant is expected to infer, for instance, the set of valid bytestreams under this specification without it being spelled out here. If the applicant thinks he must make a non-obvious (but reasonable) assumption, he should make that explicit.

2.1 The Compressed Data Format

Conceptually, the compressed data consist of a sequence of pairs of bytes C_s:

$$C_s = \{(p_0, q_0), (p_1, q_1), \ldots, (p_{n-1}, q_{n-1})\} \tag{1}$$

These pairs come in two forms; each pair can either be in the form ($p_i = 0, q_i = c$), where c is any byte value, or it can be composed of two positive integers such that $0 < q_i \leq p_i$ (with $i = 0, \ldots, n-1$). In the first case, the pair represents exactly one decoded byte (the value of q_i). In the other, the pair represents the sequence obtained by taking q_i bytes starting from an offset of p_i bytes behind in the decoded stream (see below).

2.2 Description of the Decoding Algorithm

The decoding algorithm is an implementation of a (non-injective) function that takes a sequence C_s back to its decoded form s.

Starting with an empty result bytestring, read the encoded pair sequence (with length n) from the left ($i = 0$) to the right. For each pair read, if $p_i = 0$, append q_i to the output stream. Otherwise, $p_i > 0$. Read the last p_i characters appended to the result string and take the first (from the left) q_i characters.

For example, the following sequence:

$$(0, 61), (1, 1), (0, 62), (3, 2), (3, 3) \tag{2}$$

would result in the bytestring `61 61 62 61 61 62 61 61` (represented here in hexadecimal) through the sequence [61], [61 61], [61 61 62], [61 61 62 61 61] ((3, 2) means go back three characters, in this case to the first position, and take the first two characters afterwards), and finally [61 61 62 61 61 62 61 61].

The hexadecimal values 61 and 62 become the characters `a` and `b` when interpreted as text. So the final output, when converted from bytes to text, is the string `aabaabaa`. However for the purposes of this assignment the conversion from bytes to text is out of scope and irrelevant, we will effectively be dealing with binary data only.

2.3 Compressed Data Bytestream Representation

Each pair will be represented, in order, as a sequence of two unsigned bytes. There will be no padding between the pairs. For instance, a possible encoding for the byte array `20 2A 20` would be `00 20 00 2A 02 01` (with all bytes represented here as their hexadecimal values).

2.4 Input of the Program

On input, the program will receive, on standard input, an encoded data stream. There are no restrictions on the length of input data; indeed, it can be unbounded.

2.5 Output of the Program

The program should write:

- To standard output, the decoded binary data. If any invalid or incomplete pair is found, the byte `3F` should be output in its place.
- To standard error, the result of re-encoding the decoded data produced in the above step. Write two implementations, a trivial one that always uses $p_i = 0$ and a better one that produces an output as short as you can manage.

The program should default to the latter and only switch to the first one if the environment variable `USE_TRIVIAL_IMPLEMENTATION` has value `1`.

It is left undefined how much data the program should buffer before decoding or re-encoding data.

3 Closing

You are expected to write a simple command line program that does the required I/O and implements the described algorithms. If you have insufficient time to finish the solution, please try to focus on the algorithms first. If you have time left, don't forget to add some minimal documentation and unit tests or other automatic tests.

References

[1] Taskinoor Hasan. "The Abuse of Design Patterns in Writing a Hello World Program." Internet: `http://taskinoor.wordpress.com/2011/09/21/the-abuse-of-design-patterns-in-writing-a-hello-world-program/`. Sept. 21 2011 [Aug. 29, 2013]