EN3240: Embedded Systems Engineering
Assignment 3 — Programming: Code Compression

Name: John Doe
Index No: XXXXX

August 29, 2022

This is an individual assignment!
Due Date: 25 September 2022 by 11.59 PM

Instructions

In this project, you need to implement both code compression and decompression using C, C++ or Java. Assume that the dictionary can have eight entries (index 3 bits) and the eight entries are selected based on frequency (the most frequent instruction should have index 000). If two entries have the same frequency, the priority should be given to the one that appears first in the original program order. The original code consists of 32-bit binaries. You are allowed to use only the following seven possible formats for compression. Note that if one entry (32-bit binary) can be compressed in more than one way, choose the most beneficial one i.e., the one that provides the shortest compressed pattern. If two formats produce exactly the same compression, choose the one that appears earlier in the following listing (e.g., run-length encoding appears earlier than direct matching). Also, if there is a scenario where you have two possible ways of applying bitmasks to a 32-bit binary, always give preference to the scenario where the leftmost bit '1' for the bitmask pattern (e.g., 11 is preferred over 01). Please count the starting location of a mismatch from the leftmost (MSB) bit of the pattern – If the mismatch is at the MSB, Mismatch Location should be 00000.

Format of the Run Length Encoding (RLE)

000	Run Length Encoding (2 bits)

Format of bitmask-based compression – starting location is counted from left/MSB

001	Starting Location (5 bits)	Bitmask (4 bits)	Dictionary Index (3 bits)

Format of the 1 bit Mismatch – mismatch location is counted from left/MSB

010	Mismatch Location (5 bits)	Dictionary Index (3 bits)

Format of the 2 bit consecutive mismatches – starting location is counted from left/MSB

011	Starting Location (5 bits)	Dictionary Index (3 bits)

Format of the 2 bit mismatches anywhere – Mismatch locations (ML) are counted from left/MSB

100	1st ML from left (5 bits)	2nd ML from left (5 bits)	Dictionary Index (3 bits)

Format of the *Direct Matching*

101	Dictionary Index (3 bits)

Format of the *Original Binaries*

110	Original Binary (32 bits)

Figure 1: Compression formats.

Run-Length Encoding (RLE) can be used when there is consecutive repetition of the same instruction. The first instruction of the repeated sequence will be compressed (or kept uncompressed if it is not part of the dictionary) as usual. The remaining ones will be compressed using RLE format shown above. The two bits in the RLE indicates the number of occurrences (00, 01, 10 and 11 imply 1, 2, 3 and 4 occurrences, respectively), excluding the first one. A single application of RLE can encode up to 4 instructions. Assume that the longest sequence can be at most 5 repeating instructions (the first one using other formats and the last 4 using RLE). Note that, RLE should be used when it is profitable compared to other available options.

You are expected to implement the compression and decompression functions using C, C++ or Java. You need to show a working prototype that will take any 32-bit binary (0/1 text) file and compress it to produce an output file that shows compressed patterns arranged in a sequential manner (32-bit in each line, last line padded with 1's, if needed), a separation marker "xxxx", followed by eight dictionary entries. Your program should also be able to accept a compressed file (in the above format) and decompress to generate the decompressed (original) patterns. Please see the sample files posted in the Moodle to avoid any confusion.

Command Line and Input/Output Formats:

The simulator should be executed with the following command line. Please use parameters "1" and "2" to indicate compression, and decompression, respectively.
./SIM 1 (or **java SIM 1**) for compression
./SIM 2 (or **java SIM 2**) for decompression

Please hardcode the input and output files as follows:

1. Input file for your compression function: **original.txt**
2. Output produced by your compression function: **cout.txt**
3. Input file for your decompression function: **compressed.txt**
4. Output produced by your decompression function: **dout.txt**

Submission Policy:

Please follow the submission policy outlined below. There will be up to 10% **score penalty** based on the nature of submission policy violations.

1. Please submit only one source file that includes both compression and decompression. **Please add ".txt" at the end of your filename.** Your file name must be SIM (e.g., SIM.c.txt **or** SIM.cpp.txt **or** SIM.java.txt). On top of the source file, please include the sentence: "/* On my honor, I have neither given nor received unauthorized aid on this assignment */".

2. Please test your submission. These are the exact steps that we will follow to grade your submission.
   - Download your submission from Moodle (ensures your upload was successful)
   - Remove ".txt" extension (e.g., SIM.c.txt should be renamed to SIM.c)
   - Please compile to produce an executable named SIM.
     - gcc SIM.c –o SIM **or**
     - javac SIM.java **or**
     - g++ SIM.cpp –o SIM **or**
     - g++ -std=c++0x SIM.cpp -o SIM
   - Please do not print anything on screen.
   - Assume hardcoded input/output files as outlined in the assignment description.
   - Compress the input file (original.txt) and check with the expected output (compressed.txt)
     - ./SIM 1 (or java SIM 1)
     - diff –w –B cout.txt compressed.txt
   - Decompress the input file (compressed.txt) and check with the expected output (original.txt)
     - ./SIM 2 (or java SIM 2)
     - diff –w –B dout.txt original.txt

3. *In previous years, there were many cases where output format was different, filename was different, command line arguments were different, or Moodle submission was missing. All of these led to un-necessary waste of time for TA, instructor and students.* **Please use the exactly same commands as outlined above to avoid 10% score penalty.**

4. **You are not allowed to take or give any help in completing this project.** *In previous years, some students violated academic honesty (giving help or taking help in completing this project). We were able to establish cheating in several cases - those students received "0" in the project. This time we will have* **double penalty** *for cheating.*

Grading Policy

The Moodle page has the sample input and output files. Correct handling of the sample input will be used to determine 60% of credit awarded. The remaining 40% will be determined from other input test cases that you will not have access prior to grading. The other test cases can have different types and number of 32-bit binaries (0/1 text). It is recommended that you construct your own sample input files with which to further test your compression and decompression functions. You can assume that we will use less than 128 32-bit binary (0/1 text) patterns in the new test file. **Please note that the new test case will NOT test any exceptional scenarios that are not described in this document.**