Project 2

CDA 4630/5636: Embedded Systems

Posted: February 26, 2019 Total: **10** points **Due**: April 10, 2019, 11:30 PM

In this project, you need to **implement both code compression and decompression using C, C++ or Java**. Assume that the dictionary can have sixteen entries (index 4 bits) and the sixteen entries are selected based on frequency (the most frequent instruction should have index 0000). If two entries have the same frequency, priority is 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 eight possible formats for compression (as outlined below). 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*). If a 32-bit binary can be compressed using multiple dictionary entries by any specific compression format (e.g., *bitmask-based compression*), please use the dictionary entry with the smallest index value. Please count the starting location of a mismatch from the leftmost (MSB) bit of the pattern – the position of the leftmost bit is 00000.

Format of the *Original Binaries*

000	Original Binary (32 bits)

Format of the *Run Length Encoding* (RLE)

001	Run Length Encoding (3 bits)

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

010	Starting Location (5 bits)	Bitmask (4 bits)	Dictionary Index (4 bits)

Please note that a bitmask location should be the first mismatch point from the left. In other words, the leftmost bit of the 4-bit bitmask pattern should be always '1'.

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

011	Mismatch Location (5 bits)	Dictionary Index (4 bits)

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

100	Starting Location (5 bits)	Dictionary Index (4 bits)

Format of the *4-bit consecutive mismatches* – starting location is counted from left/MSB

101	Starting Location (5 bits)	Dictionary Index (4 bits)

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

110	1st ML from left (5 bits)	2nd ML from left (5 bits)	Dictionary Index (4 bits)

Format of the *Direct Matching*

111	Dictionary Index (4 bits)

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 three bits in the RLE indicates the number of occurrences (000, 001, 010, 011, 100, 101, 110 and 111 imply 1, 2, 3, 4, 5, 6, 7 and 8 occurrences, respectively), excluding the first one. A single application of RLE can encode up to 8 instructions. Assume that the longest sequence can be at most 9 repeating instructions (the first one using other formats and the last 8 using RLE). Note that, RLE should be used when it is profitable compared to other available options.

You need to show a working prototype that will take any 32-bit binary (0/1 text) file and compress it to produce a output file that shows compressed patterns arranged in a sequential manner (32-bit in each line, last line padded with 0's, if needed), a separation marker "xxxx", followed by sixteen 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 webpage.

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. **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 following 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 we will follow too.

   - Download your submission from eLearning (ensures your upload was successful).
   - Remove ".txt" extension (e.g., SIM.c.txt should be renamed to SIM.c)

- Login to **thunder.cise.ufl.edu** or **storm.cise.ufl.edu**. If you don’t have a CISE account, go to http://cise.ufl.edu/help/account.shtml and apply for one CISE class account. Then you use **putty** and **winscp** or other tools to login and transfer files.
- 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 project 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 e-Learning submission was missing. All of these led to un-necessary frustration and 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 the previous years, some students violated academic honesty (giving help or taking help in completing this project). We were able to establish violation in several cases - those students received “0” in the project and their names were reported to UF Ethics office. This year we would also impose one additional letter grade penalty. Someone could potentially lose two letter grade points (e.g., “A-” grade to “B” grade) – one for getting 0 score in the project and then another grade point penalty on top of it. Moreover, the names of the students will also be reported to UF Dean of Students Office (DSO). If your name is already in DSO for violation in another course, the penalty for second offence is determined by DSO. In the past, two students from my class were suspended for a semester due to repeat academic honesty violation (implies deportation for international students).

Grading Policy

The class assignments webpage 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 1024 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**.