Software Engineering 265
Software Development Methods
Summer 2017

Assignment 3

Due: Thursday, July 13, 11:55 pm by "git push"
(Late submissions **not** accepted)

Programming environment

For this assignment you must ensure your work executes correctly on *linux.csc.uvic.ca*. You can use *git push* and *git pull* to move files back and forth between your account on the UVic CSC filesystem and your own computer(s). Bugs in the kind of programming done this term tend to be platform specific, and so something that works perfectly on your own machine may end up either on *linux.csc* or behaving differently because of system defaults. The actual coding fault is very rarely that of *linux.csc*'s configuration. "It worked on my machine!" will be treated as the equivalent of "Mein Schäferhund hat meine Hausaufgabe gefressen!"

Individual work

This assignment is to be completed by each individual student (i.e., no group work). Naturally you will want to discuss aspects of the problem with fellow students, and such discussion is encouraged. **However, sharing of code fragments is strictly forbidden without the express written permission of the course instructor (Zastre).** If you are still unsure regarding what is permitted or have other questions about what constitutes appropriate collaboration, please contact me as soon as possible. (Code-similarity analysis tools will be used to examine submitted work.) The URLs of significant code fragments you have found and used in your solution must be cited in comments just before where such code has been used.

Objectives of this assignment

- Revisit Python 3 for a second programming task.
- Implement the second stage of the compression scheme (move-to-front encoding with run-length encoding)
- Use *git* to track changes in your source code and annotate the evolution of your solution with "messages" provided during commits.
- Test your code against the twenty provided test-case sets.

This assignment: *phase2.py*

For this assignment you will write a Python program that reads in a phase 1 file, performs move-to-front encoding of its contents followed by run-length encoding, and then writes the result into a phase 2 files. Depending upon the phase 1 file, the phase 2 may be significantly smaller. (For more about this, please read the "Appendix" at the end of this description). Your script will also be able to read the contents of a phase 2 file, perform run-length decoding and move-to-front decoding such that the original phase 1 data is recovered.

Step 1: Move-to-front coding

Move-to-Front (MTF) Coding is an adaptive coding scheme where frequently-appearing symbols – for our assignment "symbols" equal "chars", including such char codes as \x03 and \x0a – are replaced with numbers that behave as indices. MTF acts in a manner somewhat similar to the way we might use a vertical stack of books. If we need a book from the middle of the pile, we retrieve it and when finished put the book back on top of the pile.

The figure below shows a string of input symbols ("abcabcaaaaaaaab"), the output generated by the encoding, and finally the order in which symbols appear in a list *after* each input symbol is processed. Notice the cases when a symbol is moved from a position within the word list to the top position of the list.

input	**a**	**b**	**c**	**a**	**b**	**c**	**a**	**a**	**a**	**a**	**a**	**a**	**a**	**a**	**b**
encoding	*1 a*	*2 b*	*3 c*	*1*	*2*	*3*	*3*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*3*
list after encoding	a	a b	a b c	a b c	b a c	c b a	a c b	a c b	a c b	a c b	a c b	a c b	a c b	a c b	b c b

When a new character appears in the input we (a) output the code corresponding to the first unused position in the list and (b) follow this with the character. If a character has already appeared, then we find its position in the list, output that position, and the move the character to the front of the list. A decoder can read the output and not only build the list on the fly, but also use codes (i.e., the "1 2 3 3" sequence above) to retrieve a character already in the list. (Codes represents the position of the char in the list before it is moved to the top.)

You may have noticed above that the first position in the list is numbered 1 instead of 0. There is a reason for this, and it is described in the next step.

Here is what it might look like to call *mtf_encode()* in a Python 3 program:

```
>>> s = "abcabcaaaaaaaab"
>>> mtf_encode(s)
[1, 'a', 2, 'b', 3, 'c', 1, 2, 3, 3, 1, 1, 1, 1, 1, 1, 1, 3]
>>>
```

Step 2: Run-length Encoding

Although MTF coding does not reduce the size of data, run-length encoding (RLE) can result in size reduction. In fact, it is one of the simplest forms of data compression. Quite simply, RLE replaces repeatedly-occurring symbols (e.g., chars, or integers) with a count of those symbols. If the symbol + count requires less memory to represent than original sequence of repeated symbols, then the final data size is reduced.

In principle RLE can be used with any long sequence of symbols, but for our assignment we will only deal with long sequences of 1s. (Being able to get long sequences of 1s was the whole purpose of the phase 1 transform – but you can read a bit more about this in the "Appendix" to this assignment.)

The figure below shows the MTF encoding from our example with its RLE equivalent.

MTF	*1*	*a*	*2*	*b*	*3*	*c*	*1*	*2*	*3*	*3*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*3*
RLE	*1*	*a*	*2*	*b*	*3*	*c*	*1*	*2*	*3*	*3*	*0*	*7*						*3*

The sequence of seven 1s in the MTF encoding are replaced in the RLE version with 0 followed by 7. That is, the code 0 is reserved to mean "repeated 1s" with the following number being the length of the repeat. This replacement will only ever be applied to sequences of 1s with a length of three or greater. (Now you should see why 0 is not used as an index in the MTF encoding.)

Going backwards from RLE to MTF is straightforward (i.e., replace all 0 codes with the right numbers of 1s).

Here is what it might look like to call *run_length_encode()* in a Python 3 program:

```
>>> s = "abcabcaaaaaaaab"
>>> e = mtf_encode(s)
>>> run_length_encode(e)
[1, 'a', 2, 'b', 3, 'c', 1, 2, 3, 3, 0, 7, 3]
```

Step 3: Using ASCII to represent the encoding

All that is left is to write the phase 2 file. Phase 2 files will have their own magic number ('\xda', '\xaa', '\xaa', and '\xad') and the blocksize stored in the phase 1 file

will be stored in the phase 2 file; in that way when we go backwards from a phase 2 file to a phase 1 file, we can recover the blocksize needed for the phase 1 data.

What we will do is write to the phase 2 file each element in the RLE list using the ASCII code. This is trivial for characters such as 'a' (i.e., ascii code 97 or \x61) but is a bit less obvious how to do this for numbers such as 0, 1, or 7.

Therefore we will make the following assumptions for this assignment:

- As in phase 1, phase 2 files will be written in Python 3 using the "latin-1" character encoding.
- In a phase 1 file, other than the magic number and block size, all remaining data will be ASCII characters whose values range from 0 up to 127 (i.e., \x7f). That is, there are no ASCII characters with values from 128 (\x80) to 255 (\xff). This assumption is realistic for us as our phase 1 transform is only applied to text data.
- We can use ASCII codes from 128 to 255 to represent numbers in our encoding. In essence, we take the number, add 128 to it, and get our ASCII code. For example, the number 7 will become ASCII 135 (\x87).
- The MTF representation will never produce a code greater than 127.

Here is what a call to *into_ascii()* would look like:

```
>>> s = "abcabcaaaaaaaab"
>>> e = mtf_encode(s)
>>> r = run_length_encode(e)
>>> a = into_ascii(r)
['\x81', 'a', '\x82', 'b', '\x83', 'c', '\x81', '\x82', '\x83', '\x83', '\x80',
'\x87', '\x83']

>>> "".join(a)
'\x81a\x82b\x83c\x81\x82\x83\x83\x80\x87\x83'
```

Note the string produced above from the call to *join()* can be used as an argument to some call to *write()* for a file.

There is one wee "gotcha" for *into_ascii()*, however. As Python is dynamically typed, it permits lists to consist of a mix of types. The output from *run_length_encode* is a list made up of integers and strings of length one (i.e., chars). We only want to add 128 to those list elements which are integers, and leave the strings/chars untouched.

We can solve this by making use the Python built-in function *isinstance()* as part of our loops and conditionals. Here are some examples of calls to *isinstance()*:

```
>>> m = 3
>>> n = "foo"
>>> isinstance(m, int)
True
>>> isinstance(n, int)
False
```

(For further thought: How could we handle run-lengths greater than 127?)

Step 4: Putting it all together

Unlike the phase 1 algorithm, this phase 2 step is able to treat everything in the phase 1 file from char 9 onwards as one long string. Therefore the phase 2 algorithm will be:

a. Open phase 1 file.
b. Read first four chars, and verify they are from a phase 1 file.
c. Read next four chars into a variable (i.e., contains the block size)
d. Read in remaining chars as one long string.
e. Close phase 1 file.
f. Open new phase 2 file.
g. Write phase 2 magic number (four chars)
h. Write the chars we saved earlier (from step c.)
i. Perform *mtf_encode* on the one long string (from step d.)
j. Perform *run_length_encode* on the previous result (from step i.)
k. Perform *into_ascii* on the previous result (from step j.)
l. Write the string corresponding step k's result to the open file.
m. Close phase 2 file.

```
% ./phase2.py --encode --infile ~zastre/seng265/tests/t01.ph1 --outfile t01.ph2
% hd t01.ph2
00000000  da aa aa ad 14 00 00 00  81 0a 82 6c 83 74 84 70  |⇜??.......l.t.p|
00000010  82 85 61 85 86 03 81                              |..a....|
00000017
```

Going from phase 2 to phase 1 is just a reversal of steps.

a. Open phase 2 file.
b. Read in first four chars, and verify they are from a phase 2 file.
c. Read next four chars into a variable (i.e., contains block size to be stored in phase 1 file).
d. Read in remaining chars as one long string.
e. Close phase 2 file.
f. Open new phase 1 file.
g. Write phrase 1 magic number (four chars)
h. Write the chars we saved earlier (from step c.)
i. Perform the reverse of *into_ascii* on the one long string (from step d.)
j. Perform run_length_decode on the previous result (from step i.)
k. Perform mtf_decode on the previous result (from step j.)
l. Write the string corresponding to step k's result to the open file.
m. Close phase 1 file.

```
% ./phase2.py --decode --infile t01.ph2 --outfile t01.ph1
% diff ~zastre/seng265/tests/t01.ph1 t01.ph1
% # :-)
```

Note that steps b. above might seem a little silly (i.e., checking for magic numbers), but having these checks in place there will save a lot of heartburn especially as the number of little test files begins to multiply. I write from experience.

Exercises for this assignment

1. Write your program *phase2.py* program in the *a3/* directory within your *git* repository.
   - Examine command-line arguments to determine what operations, and which file, the program will be processing. Ensure your usage matches the examples shown earlier (i.e., use "--encode" and "--decode", not "--forward" or "--backward".

2. You are permitted to use more than one file for your submission, and even write your own class(es) if you wish. However, make sure all files that make up your solution are in the git repo.

3. **Code restrictions: Use the script structure described in lectures (i.e., all code is defined in functions, with the script involving the *main()* function via an *if* statement). The only global variables permitted are those for constants (i.e., the magic number, the end-of-block symbol). Do not write your own Python classes.**

4. Use the test files in */home/zastre/seng265/tests* to guide your implementation effort. Start with simple cases (for example, the one described in this writeup). In general, lower-numbered tests are simpler than higher-numbered tests. Refrain from writing the program all at once, and budget time to anticipate for when "things go wrong". There are ten pairs of test files.

5. For this assignment you can assume all test inputs will be well-formed (i.e., the teaching team will not test your submission for handling of errors in the input). Later assignments will specify error-handling as part of the assignment.

6. Use *git add* and *git commit* appropriately. While you are not required to use *git push* during the development of your program, you **must** use *git push* in order to submit your assignment.

What you must submit

- A Python script named *phase2.py* plus any other needed Python files are added and committed within your git repository as these will be your

solution to Assignment #3. Ensure your work is **committed** to your local repository **and pushed** to the remote **before the due date/time**.

Evaluation

Students will demonstrate their work to a member of the course's teaching team. Sign-up sheets for demos will be provided a few days before the due-date; each demo will require around 10 minutes.

Our grading scheme is relatively simple.

- "A" grade: An exceptional submission demonstrating creativity and initiative. *phase2.py* runs without any problems. All twenty sets of tests pass. The program is clearly written and uses functions appropriately (i.e., is well structured).
- "B" grade: A submission completing the requirements of the assignment. *phase2.py* runs without any problems; all twenty sets of tests pass. The program is clearly written.
- "C" grade: A submission completing most of the requirements of the assignment. *phase2.py* runs with some problems; some tests do not pass.
- "D" grade: A serious attempt at completing requirements for the assignment. *phase2.py* runs with quite a few problems; most tests do not pass.
- "F" grade: Either no submission given, or submission represents very little work.

Appendix

Some of you will observe the data compression results achieved as being rather underwhelming. For example:

- Test 01: t01.ph1 = 17 bytes; t01.ph2 = 23 bytes
- Test 08: t08.ph1 = 1010 bytes; t08.ph2 = 1044 bytes
- Test 20: t20.ph1 = 53364; t20.ph2 = 48221 (i.e., finally a smaller ph2 file!)
- In all test cases, the phase 2 file is still larger than the original text file!

These observations (if made!) are good ones. Have a look, however, at the following table involving some experiments with test 20's data. The table shows the size of the phase 2 file when different blocksizes are used to generate the phase 1. (The original size of t20.txt is 53249 bytes.)

Block size	Phase 1 size (bytes)	Phase 2 size (bytes)
1000	53311	46644
2000	53284	45702
5000	53268	43121

As the block size increases, the phase 2 file size decreases. This makes sense. The phase 1 transform (also known as the Burrows-Wheeler transform) results in longer and longer repeated sequences of characters within the same block. So the block size increases, so do the lengths of repeated chars, and therefore MTF + RLE is able to better reduce data size. For example, when generate the phase 2 file for the last table row, some of the runlengths of 1st in the MTF encoding are quite large (i.e., over 500).

In fact, the *bzip2* compression algorithm uses, amongst other things, the Burrows-Wheeler transform. For bzip2 the default block size is 900,000 bytes (!!!) and the program also works with binary data. (Our work so far this semester has been implementing some bits and pieces of *bzip2*.) It is far beyond the scope of this course to fully implement *bzip2*, but the bits and pieces we can implement are very instructive.