Software Engineering 265
Software Development Methods
Fall 2018

Assignment 4

Due: Wednesday, December 05, 11:55 pm by "git push"
(Late submissions **not** accepted)

Programming environment

For this assignment you must ensure your work executes correctly on the Linux machines in ELW B238. You are free to do some of your programming on your own computers; if you do this, give yourself a few days before the due date to iron out any bugs in the C program you have uploaded to the lab machines.

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.** 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 may be used to examine submitted programs.)

Objectives of this assignment

- Revisit the C programming language, this time using dynamic memory allocation.
- Learn and implement Move-to-front (MTF) and Run-length encoding (RLE) algorithms that are popularly used in the compression scheme.
- Use *git* to track changes in your source code and annotate the evolution of your solution with "messages" provided during commits.
- Test your code with the built-in testing functions (No text input files).
- Be prepared to justify your script's **handling of errors** for this assignment.

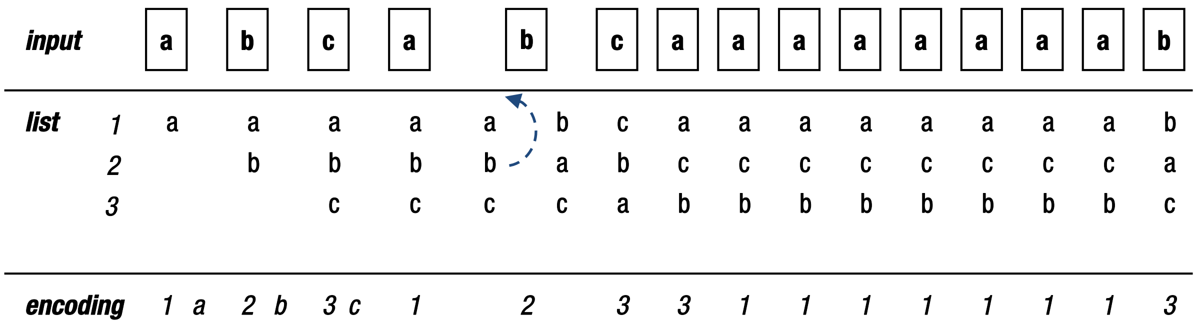
This assignment: *sengencode.c*

For this assignment you will write a C program that can perform move-to-front encoding of its contents followed by run-length encoding. **In essence, you will be implementing run-length encoding using dynamic memory in C, the implementation of MTF encoding algorithm has been given to you in order to help you complete the rest of your C program.**

Step 1: Understand 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
list
1 a a a a a b c a a a a a a a a b
2 b b b b a b c c c c c c c c a
3 c c c c a b b b b b b b b c
encoding
1 a 2 b 3 c 1 2 3 3 1 1 1 1 1 1 1 3

In this assignment, we will use an initial static large table from ASCII to index up to 256 characters. However, in a more general case, you can improve the given MTF algorithm to be adaptive to any new characters (e.g. characters from Unicode), but this is **NOT** required in this assignment. For example, when a new character appears in the input we can (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 of implementing RLE algorithm.

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

```
…

char s[] = "abcabcaaaaaaaab";

int len = strlen(s);

int *code = (int *) malloc(sizeof(int) * len);

mtf_encode(s, len, code);

print_mtf_encode(code);

…

---------------------------------------------------------
$> gcc mtfencode.c –o mtfencode

$> ./mtfencode

[1, 2, 3, 1, 2, 3, 3, 1, 1, 1, 1, 1, 1, 1, 3]
```

Note:

The above result is based on the general case which uses a **dynamic table** to index the characters. This is the reason why the letter ‘a’ has the index 1. However, in the given ***sengencode.c*** file, you will see a larger index number for the letter ‘a’ because the **static size table** was created from ASCII and ‘a’ is in the middle of the table.

You should notice the MTF encoding is reversible. Simply maintain the same initial table and decode by replacing each index number in the encoded stream with the letter at that index in the list.

Assuming we have an initial table as “abcdefghijklmnopqrstuvwxyz”. You take the number "2" of the encoded block and look it up in the list, which results in "b". Then move the "b" to front which results in (bacdef...). Then take the next "2", look it up in the list, this results in "a", move the "a" to front… etc.

Step 2: Implement Run-length Encoding and Decoding

Although MTF encoding 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 another story with applying the famous Burrows–Wheeler transform, you can learn more details by searching online if you have time.)

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

MTF	*1*	*2*	*3*	*1*	*2*	*3*	*3*	*1*	*1*	*1*	*1*	*1*	*1*	*1*	*3*
RLE	*1*	*2*	*3*	*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). The **three** left-blank functions in the C program should be your target in this assignment.

Exercises for this assignment

1. Write your program *sengencode.c* program in the *A4/* directory within your *git* repository.

2. The tester function run_length_test() is already given in the C program. You can use it to test and debug your program. Refrain from writing the program all at once, and budget time to anticipate for when "things go wrong". There are 5 built-in testing cases.

3. 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.

4. Reasonable run-time performance of your script is expected. None of the test cases should take longer than **15 seconds** to complete on a machine in ELW B238.

What you must submit

- A C program named *sengencode.c* should be added and committed within your git repository as it will be your solution to Assignment #4. Ensure your work is **committed** to your local repository **and pushed** to the remote **before the due date/time**.
- A text file named "error_handling.txt" which enumerates the errors that are addressed by your submission.

Evaluation

Our grading scheme is relatively simple.

Requirement	**Marks**
File "sengencode.c" compiles and runs without errors.	1
The program is clearly written and uses **functions** appropriately (i.e., is well structured).	1
Errors have been enumerated and are either suitably handled or a sensible response strategy has been suggested.	2
Code passes the RLE encoding in the 5 built-in test cases	3
Code passes the RLE decoding and recovers the original messages in the 5 built-in test cases	3
Total	**10**