Software Engineering 265
Software Development Methods
Summer 2017

Assignment 1

Due: Wednesday, ~~June 6~~ June 7, 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 computer. Bugs in the kind of programming done this term tend to be platform specific, and so omething that works perfectly on your own machine may end up crashing on *linux.csc*. The fault is very rarely that of *linux.csc*'s configuration. "It worked on my machine!" will be treated as the equivalent of "The dog ate my homework!"

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

- Understand a problem description, along with the role played by sample input and output for providing such a description.
- Use the C programming language to write the first phase of data compressor. The program will be named (rather unimaginatively) *phase1.c*.
- 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 ten provided test cases.

This assignment: *phase1.c*

The sum total of this semester's assignments will be code for a lossless data compressor. The complete data compressor will combine three distinct processing phases. This assignment tackles the first phase.

In a nutshell, data compression algorithms attempt to exploit *data redundancy* within a file in order to create an encoding of the file smaller than the original. For example, a sequence of repeated characters can be replaced by one instance of the character followed by a code indicating the number of repeats (i.e., *run-length encoding*). The character plus code can often be smaller (sometimes *much* smaller) than the original sequence of characters. Lossless data compressors (such as Unix *compress*, *zip*, *gzip*, *bzip2*, etc.) ensure the original file can always be recovered from the smaller encoding.

Some data compression algorithms apply a first step of transforming the original file in such a way as to increase redundancy before applying compression. Such a transform is what our *phase1.c* will accomplish.

Consider the following string that is eight characters long:

p	a	l	a	t	a	l	•

It consists of seven alphabetic characters plus a special "end-of-text" symbol which is represented here as a dot (but in reality is ASCII code 3). Note that this string itself must be stored in a character array with at least nine chars, i.e., the actual end of the string within a char array is represented in C by the NUL character or ASCII code 0. (To learn more about ASCII, enter *man ascii* at the command shell.)

The forward direction of our phase-one transform to the string produces:

l	t	p	l	a	a	•	a

(This example is the first test case – *t01.txt*, *t01.ph1* – amongst the test files provided to you in */home/zastre/seng265/tests*.) Note some repeated characters (but not all of them) are somewhat "bunched up" in this second string.

The algorithm for the forward direction of the transform is relatively simple. *Rotated copies* of the original string are stored in a two-dimensional character array; the rows of the array are then sorted in *lexicographic* (i.e., *dictionary*) order; and finally the last column of the two-dimensional array is the phase-one transform (i.e., we'll need to copy this column's data into some other character array). Here is a two-dimensional array of the original string with its rotations but before sorting:

p	a	l	a	t	a	l	•
a	l	a	t	a	l	•	p
l	a	t	a	l	•	p	a
a	t	a	l	•	p	a	l
t	a	l	•	p	a	l	a
a	l	•	p	a	l	a	t
l	•	p	a	l	a	t	a
•	p	a	l	a	t	a	l

And here are the array's contents after sorting the rows. I have highlighted the last column. (In lexicographic order, the "end-of-text" symbol comes before all other alphabetic characters.)

•	p	a	l	a	t	a	l
a	l	•	p	a	l	a	t
a	l	a	t	a	l	•	p
a	t	a	l	•	p	a	l
l	•	p	a	l	a	t	a
l	a	t	a	l	•	p	a
p	a	l	a	t	a	l	•
t	a	l	•	p	a	l	a

And how do we obtain the original string from the phase-one string, i.e., what is the backward direction to the transform? Going backwards is harder than going forward but not in a complicated way.

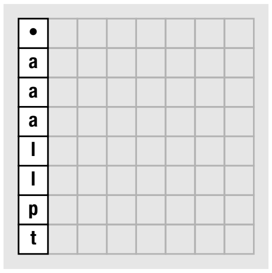
Recall our phase-one string:

l	t	p	l	a	a	•	a

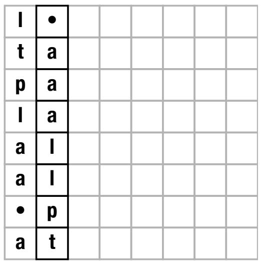
In step 1 of the algorithm, we start the backwards direction by copying the string into the first column of a 2D array whose height (i.e., # of rows) equals the length of the string:

l							
t							
p							
l							
a							
a							
•							
a							

For step 2, we then copy the contents of the *original* 2D array into a *temporary* 2D array and sort the rows of the latter in lexicographic order:

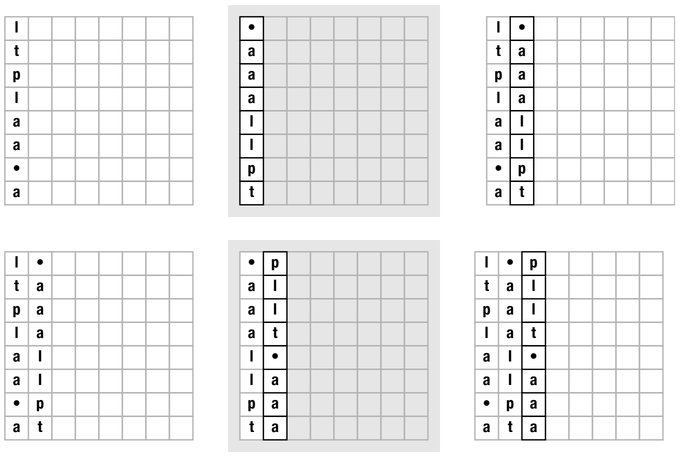


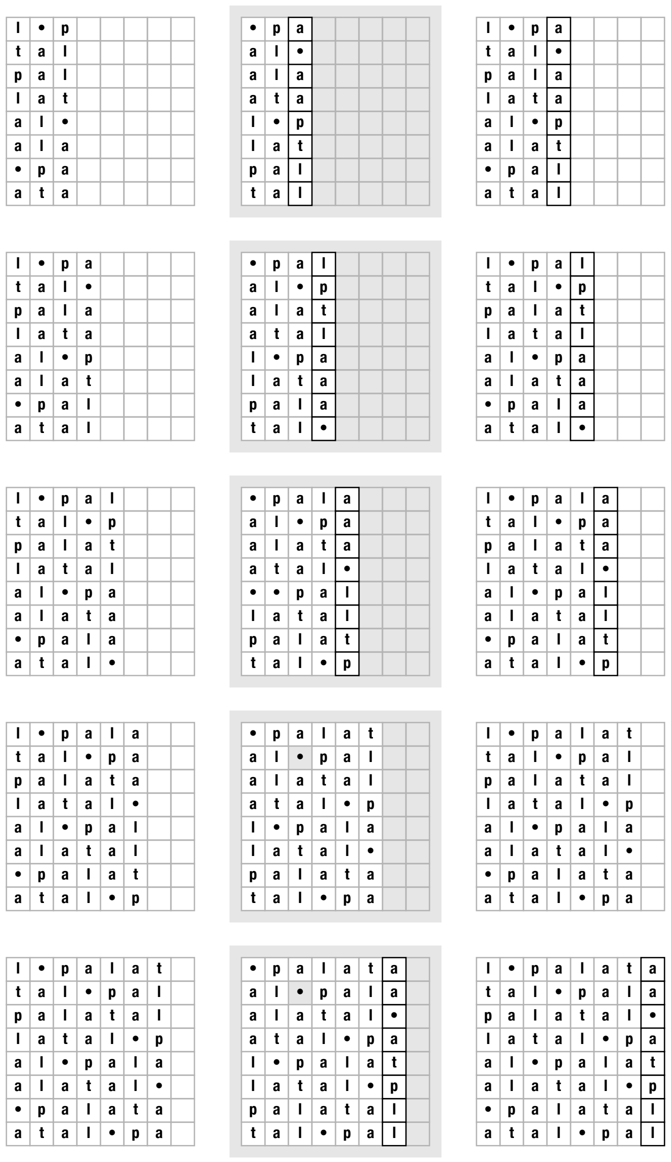
Finally, in step 3 the column from the *temporary* 2D array (actually the right-most column with text) is then copied back into the *original* 2D array:



(The bold-faced lines around columns/cells are meant to emphasize the relationship between data in the sorted array and what is copied back into the original array.)

Steps 2 and 3 are repeated until the original array no longer has empty columns. The original string will be at the row having the end-of-text symbol in the last column. What follows is a diagram of the process from start-to-end for our example. Each row in the diagram shows three arrays: (a) the original array before the step is taken; (b) the temporary array after copying and sorting; and (c) the original array after copying the right-most column from the sorted temporary array.





To find the final result of the backwards direction of the transform string, we look at each row of the final array and choose the one where the last character is the end-of-text symbol. In our example the final 2D array has the input string in the third row, and therefore this is the string corresponding to the results of backwards direction of the transform (i.e., the original string we used for the forward direction).

Blocks

While the above example involves a single word in a single string, we want to apply our transform on larger amounts of text. (In the next section below is a description of how we'll store phase-one results.) In this first assignment we will work with 20-character *blocks* (i.e., a block is just an array) and then transform each of these in turn. (Note that a block may contain more than one C string.) We will transform a file, block by block, from the beginning to the end. If we come to the end of processing a file and find there are fewer than 20 characters remaining, then we'll simply process the remainder of the file with a block having the same size as the number of characters remaining in the file. Note, though, that we must always add the "end-of-text" symbol to our string before performing the forward direction of the transform. Therefore our scheme will not work if ASCII 3 is a legitimate symbol in the input. Each string/block resulting from the forward-direction of the transform will be therefore be 21 characters (i.e., 20 + 1) long. (For the sticklers out there, I realize the size of our output gets large with this transform, but in subsequent assignments we see how all phases can result in a net decrease in size.)

When applying the backwards direction of the transform, the code will simply read in 21-character blocks, one at a time, backwards-transforming each in turn (and producing 20-character blocks as a result). The final block may be less than 21 characters, in which case we treat this as producing an original string less than 20 characters. The last character in each block will be the ASCII 3 symbol, and we will remove this before outputting the result.

Storing results of the phase-one transform

The transformed strings will be stored in a PH1 file – this is a file format invented for the assignment. In a PH1 file:

- The first four bytes/characters are hex values ~~'\xab', '\xba', '\xfe', and '\xfe'~~ '\xab', '\xba', '\xbe', and '\xef';
- The next four bytes represent the block size use for transforms (i.e., for this assignment the bytes are the binary representation of the integer 20);
- The remainder of the file is the transformed blocks (i.e., sequences of 21 chars, with the final sequence possibly less than 21 chars in length).

For example, below is a *hexdump* showing the contents of */home/zastre/seng265/tests/t01.ph1* (which would result from the forward transform of this writeup's example). You will get to know the *hd* command very well this semester.

```
% hd /home/zastre/seng265/tests/t01.ph1
00000000  ab ba be ef 14 00 00 00  0a 6c 74 70 6c 61 61 03  |????.....ltplaa.|
00000010  61                                                |a|
00000011
```

Running the program

The direction of the transform (*--forward* or *--backward*), plus the input (*--infile*) and output (*--outfile*) filenames, must be provided at the command line. Here are some example invocations (assuming *phase1* is compiled and in the current directory):

```
% ./phase1 --forward --infile ~zastre/seng265/tests/t01.txt --outfile a.ph1
```

Here I'm checking if the output is correct by comparing it with the expected output:

```
% diff a.ph1 ~zastre/seng265/tests/t01.ph1
```

And now I am testing my program to see if it can correctly perform the backwards direction of the transform:

```
% ./phase1 --backward --infile ~zastre/seng265/tests/t01.ph1 --outfile a.txt

% diff a.txt ~zastre/seng265/tests/t01.txt
```

In both uses of *diff*, I would know the files are identical if *diff* itself prints no messages.

Exercises for this assignment

1. Write your program *phase1.c* program in the a1 directory within your git repository.
   - Examine command-line arguments to determine what operations, and which file, the program will be processing.
   - Open a file for input.
   - Read text input from a file, block by block.
   - Implement the forward and backward direction of the transform as described earlier.
   - Open a file for output, write PH1 header info, write blocks/strings to the file, and then close the file.
   - Create and use arrays in a non-trivial array.
   - Use the "-ansi" flag when compiling to ensure your code is ANSI compliant.

2. **Do not use *malloc*, *calloc* or any of the dynamic memory functions. Pathnames provided for input or output filenames will not be any longer than 60 characters. Input blocks will be no longer than 20 characters (not counting the "end-of-text" symbol).**

3. Keep all of your code in one file for this assignment. In later assignments we will use the separable compilation features of C.

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 single C source file named *phase1.c* within your git repository containing a solution to Assignment #1. Ensure your work is **committed** to your local repository **and pushed** to the remote **before the due date/time**. (You may keep extra files used during development within the repository.)
- No dynamic memory-allocation routines are permitted for Assignment #1 (i.e., do not use anything in the *malloc* family of functions).

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. *phase1* runs without any problems. All ten 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. *phase1* runs without any problems; all ten tests pass. The program is clearly written.
- "C" grade: A submission completing most of the requirements of the assignment. *phase1* runs with some problems; some tests do not pass.
- "D" grade: A serious attempt at completing requirements for the assignment. *phase1* runs with quite a few problems; most tests do not pass.
- "F" grade: Either no submission given, or submission represents very little work.