CSC 209 Assignment 1, Winter 2016

Due by the end of Friday February 5, 2016; no late assignments without written explanation.

Please re-read the statement about academic offences at the end of the course info sheet!

This assignment consists of writing five small programs. Describing five different programs makes this handout long; but I think that the actual assignment length is actually somewhat short, so don't be dismayed by the page count (which also includes a lot of explanation).

1. A shell program to add numbers

For this first part, you will write a simple shell script to add a bunch of integers. The integers may be specified on stdin or in files specified on the command-line. (They may not be specified directly on the command-line; command-line arguments will be deemed to be files, not values.)

To process zero-or-more command-line arguments in the standard way, simply use

```
cat $*
```

(If there are no command-line arguments, this expands to simply "cat", which will just copy the standard input!)
(This will run into trouble with file names with spaces in them, but you don't have to solve that for this assignment. The solution for this is discussed in the Q&A file if you are interested.)

This should be piped into a *sh* tr command to canonicalize the numbers into one number per line.

The output of that can then be piped into a 'while' loop beginning with "`while read x`" (for any variable name 'x'). See a starter version of this loop in /student/csc209/a1/starter/add

Example uses, where '$' is the shell prompt and where 'file' contains the numbers 3, 4, and 6 (separated by any white space and/or on separate lines):

```
$ echo 3 4 | sh add
7
$ sh add file
13
$ sh add /dev/null
0
$
```

2. A C program to implement a rotation code

A simple encryption algorithm is to move down the alphabet a specified number of letters, traditionally three. So 'a' encrypts as 'd', 'b' encrypts as 'e', and so on. It "wraps around" at the end of the alphabet (e.g. 'y' encrypts as 'b'), and is thus called a "rotation".

Write a C program which takes a rotation value as the sole command-line argument, and applies a rotation to its standard input. So "rot 3" would change 'a' to 'd', but "rot 4" would change 'a' to 'e'. Case should be preserved, so for example if you run "rot 3" then 'a' becomes 'd' but 'A' becomes 'D'. Non-alphabetic characters (including space and newline) will be output unmodified.

The algorithm is that if you have a letter, add the rotational value to the character value, and then while your result exceeds `'z'` or `'Z'` as applicable, subtract 26. (This is a 'while' rather than an 'if' so that commands such as "rot 27" also work.)

Note that if you encrypt by rotating *n* places, you can decrypt by rotating (26–*n*) places.

Please see "starter code" in /student/csc209/a1/starter/rot.c which takes care of the usage check and the conversion of the command-line argument to integer. And you can experiment with a compiled solution in /student/csc209/a1/rot .

(continued)

3. A shell program to help decrypt rotation cryptograms

A "cryptogram" is a puzzle where you get encrypted text and the challenge is to determine the original text.

Cryptograms involving a simple rotation code as in question 2 are easy to solve, but now you will write a shell script which makes them *trivial* to solve.

Your shell script will help decrypt its command-line arguments by outputting them rotated by all amounts from 1 to 25 inclusive; the user can then scan those 25 lines and quickly find the correct decrypted text.

Example usage: `sh decrypt Ymnx nx fs jcywjrjqd xjhwjy rjxxflj.`

To run "rot", your shell program will have to hard-code a path name for this command. Please use /student/csc209/a1/rot. But you should put this value in a variable, near the beginning of your script, so that it is easily changed when moving these programs to a different system.

4. A C program to extract whitespace-delimited fields

The "nth" command outputs the nth field of each line (one-origin), where fields are delimited by any amount of whitespace. (This differs from the behaviour of *cut*, which expects fields to be separated by a single delimiter character.) If *n* is greater than the number of fields on a particular line, it prints a blank output line for that line.

Ideally, this program should take zero or more file names in the usual way, but to simplify this assignment it will take just a single command-line argument which is the 'n' value, and will always then process its standard input. Thus you can use the rot.c starter code again usefully here,

While it would be possible to read the input file line by line and break it up into "tokens" and so forth, a more efficient algorithm (which you must use for this assignment) is based on an augmented state machine and goes as follows:

- There are integer variables 'state' and 'i'. We start at state 0 (i.e. initialize 'state' to zero before the loop). Also initialize 'i' to 1. *i* represents which field we're on within the line.
- You move to the next state based on the current state and current character (in *c*), possibly changing the value of *i*, and possibly producing output, as stated below.
- In state 0: if *c* is a space or a tab, stay in state zero; if a newline, output a newline, set *i* to 1, and stay in state zero; else if *i*<*n* then go to state 1; else output the character and go to state 2.
- In state 1: if *c* is a space or a tab, increment *i* (i.e. *i*=*i*+1) and go to state zero; if a newline, output a newline, set *i* to 1, and go to state zero; else stay in state 1.
- In state 2: if *c* is a space or a tab, go to state 3; if a newline, output a newline, set *i* to 1, and go to state 0; else output the character and stay in state 2.
- In state 3: if *c* is a newline, output a newline, set *i* to 1, and go to state 0; else stay in state 3.

Example runs:
If the input consists of the two lines

```
          Hello, world, how are you
      today?  Myself, I am a C program with no feelings.
```

then "nth 4" would produce the output

```
      are
      am
```

(without indentation) and "nth 7" would produce a blank line and then the line "`program`".

Again, please try a compiled solution in /student/csc209/a1/nth .

(continued)

5. A shell program to help find palindromes

A "palindrome" is a coherent phrase which consists of the same letters forwards and backwards, although the spacing and capitalization may be different. A C program to check whether a string is a palindrome is the subject of part three of tutorial six, and there are some example palindromes on that tutorial page. This program exits with exit status 0 for a palindrome or exit status 1 for a non-palindrome, ideal for testing in an *sh* 'if' statement.

Given that C program, write a shell script to *find* palindromes, in the following case. The arguments to your shell script are a bunch of words (as separate arguments). Some permutations of these words might constitute a palindrome. Your script will output all such permutations, if any.

Examples, where '$' is the shell prompt:

```
$ sh findpal I\'m Madam Adam
Madam I'm Adam
$ sh findpal Madam Mad
$ sh findpal Madam madam
Madam madam
madam Madam
$
```

You can find a compiled version of the "ispalindrome" program (see the tutorial) in /student/csc209/a1/ispalindrome (compiled so that you can't see a tutorial solution before its due date!), and you can find another helpful program in /student/csc209/a1/permute which produces all permutations of its argument words. (The output from *permute* can be piped into a 'while' loop as in the *add* program in question 1.)

These path names will have to be hard-coded into your shell script to make it work, but they should be in variables at the top (or the variable could just be the "/student/csc209/a1" part) so that it is easily changed.

To submit

I suggest you begin by making a new directory to hold your assignment files, plus any other working copies and associated files. You should call your assignment files add, rot.c, decrypt, nth.c, and findpal.

Once you are satisfied with your files, you can submit them for grading with the command

```
/student/csc209/submit a1 add rot.c decrypt nth.c findpal
```

You may still change your files and resubmit them any time up to the due time. You can check that your assignment has been submitted with the command

```
/student/csc209/submit -l a1
```

You can also submit files individually instead of all at once.
This is the only submission method; you do not turn in any paper.

Other notes

Note carefully the specified semantics of the command-line arguments. In *add*, the command-line arguments are file names containing the data; in the other programs, the command-line arguments are the data itself.

For now, we will run our shell scripts by typing "sh file" or "sh file args ...".

Your programs must run on the UTM linux machines, and they should use standard *sh* and C features only. For the *sh* scripts, avoid *bash* extensions, and please test them in different shells, not only in *bash*. E.g. also try "dash findpal foo".

(continued)

Do not use awk, perl, python, or any other programming language in your shell scripts besides *sh*. (These are valuable languages to know, but are not the point of questions 1, 3, and 5 in the current assignment, in which you will be graded on your *sh* programming.)

Please see the assignment Q&A web page at

`https://cs.utm.utoronto.ca/~ajr/209/a1/qna.html`

for other reminders, and answers to common questions.

Remember:

This assignment is due at the end of Friday, February 5, by midnight. Late assignments are not ordinarily accepted and *always* require a written explanation. If you are not finished your assignment by the submission deadline, you should just submit what you have (for partial marks).

Despite the above, I'd like to be clear that if there *is* a legitimate reason for lateness, please do submit your assignment late and send me that written explanation. (For medical notes, I need to see the original, in person. E.g. in office hours or after the next class.)

I'd also like to point out that even a zero out of 8% is far better than cheating and suffering an academic penalty. Don't cheat even if you're under pressure. Whatever the penalty eventually applied for cheating, it will be worse than merely a zero on the assignment. Do not look at other students' assignments, and do not show your assignment (complete or partial) to other students. Collaborate with other students only on material which is not being submitted for course credit.