CS246—Assignment 1 (Fall 2022)

R. Evans E. Lee B. Lushman

Due Date 1: Friday, September 23, 5pm
Due Date 2: Friday, September 30, 5pm

Questions 1 and 2 are due on Due Date 1;

the remainder of the assignment is due on Due Date 2.

1. Provide a Linux command line to accomplish each of the following tasks. Your answer in each subquestion should consist of a single command or pipeline of commands, with no separating semicolons (;). (Please verify before submitting that your solution consists of a single line. Use `wc` for this.) Before beginning this question, familiarize yourself with the commands outlined on the Linux handout. Keep in mind that some commands have options not listed on the sheet, so you may need to examine some man pages. Note that some tasks refer to a file `myfile.txt`. No `myfile.txt` is given. You should create your own for testing.

   (a) Print the 10th through 25th words (including the 10th and 25th words) in `/usr/share/dict/words`. You may take advantage of the fact that the words in this file are each on a separate line. Place your command pipeline in the file `a1q1a.txt`.

   (b) Print the (non-hidden) contents of the current directory in reverse of the normal order. Place your command pipeline in the file `a1q1b.txt`.

   (c) Print the number of lines in the text file `myfile.txt` that do *not* contain the string `cs246` (all in lower-case). Place your command pipeline in the file `a1q1c.txt`.

   (d) Print the first line that contains the string `cs246` (all in lower-case) from the text file `myfile.txt`. Place your command pipeline in the file `a1q1d.txt`.

   (e) Print the number of lines in the text file `myfile.txt` that contain the string `linux.student.cs.uwaterloo.ca` where each letter could be either upper-case or lower-case. (Hint: this is not as obvious as you may think–in addition to printing all lines that it should print, make sure that you are testing that it is *not* printing lines that it *shouldn't* print.) Place your command pipeline in the file `a1q1e.txt`.

   (f) Print all (non-hidden) files/directories in any *subdirectory* of the current directory that end with lower-case `.c` (immediate subdirectories only, not subdirectories of subdirectories). Do not use `find`. (Hint: there's an easy way to do this using only `ls` if you're creative with globbing patterns.) Place your command pipeline in the file `a1q1f.txt`.

   (g) Out of the first 20 lines of `myfile.txt`, how many contain at least one digit? Place the command pipeline that prints this number in the file `a1q1g.txt`.

   (h) Print all (non-hidden) files in the current directory that start with `a`, contain at least one `b`, and end with `.c` (these required letters must be in lower-case in order to match). Place your command pipeline in the file `a1q1h.txt`.

   (i) Print a listing, in long form, of all non-hidden entries (files, directories, etc.) in the current directory that are executable by at least one of owner, group, other (the other permission bits could be anything). Do not attempt to solve this problem with `find`. Place your command pipeline in the file `a1q1i.txt`.

(j) Before attempting this subquestion, do some reading (either skim the man page or have a look on the Web) on the `awk` utility. In particular, be sure you understand the effect of the command

```
awk '{print $1}' < myfile.txt
```

Give a Linux pipeline that gives a sorted, duplicate-free list of userids currently signed on to the (school) machine the command is running on.
Place your command pipeline in the file `a1q1j.txt`.

2. For each of the following text search criteria, provide a regular expression that matches the criterion, suitable for use with `egrep`. Your answer in each case should be a text file that contains just the regular expression (i.e., you don't need to include the `egrep` command in your submitted solution), on a single line (again, use `wc` to verify this). If your pattern contains special characters, enclose it in quotes.

   (a) Lines that contain both `cs246` and `cs247`, in lower-case.
   Place your answer in the file `a1q2a.txt`.

   (b) Lines that contain nothing but a single occurrence of laughter, where laughter is defined as a string of the form `Hahahahahahahahahahahahahaha!`, with arbitrarily many `ha`'s. The string must start with `H` and end with `!`. Place your answer in the file `a1q2b.txt`.

   (c) Lines that contain nothing but a single occurrence of generalized laughter, which is like ordinary laughter, except that there can be arbitrarily many (but at least one) `a`'s between each pair of consecutive `h`'s. (For example: `Haahahaaaa!`) Place your answer in the file `a1q2c.txt`.

   (d) Lines that contain at least one lower-case `a` and at least two lower-case `b`'s.
   Place your answer in the file `a1q2d.txt`.

   (e) Lines consisting of a definition of a single C variable of type `int`, without initialization, optionally preceded by `unsigned`, and optionally followed by any single line `//` comment.
   Example:

   ```
   int  varname;  //  optional comment
   ```

   You may assume that all of the whitespace in the line consists of space characters (no tabs). You may also assume that `varname` will not be a C keyword (i.e., you do not have to try to check for this with your regular expression). If you don't remember the rules for naming a C variable, please consult `https://www.programiz.com/c-programming/c-variables-constant`
   Place your answer in the file `a1q2e.txt`.

3. Write a `bash` script called `sumOfProducts` that is invoked as follows:

   ```
   ./sumOfProducts file-1 file-2 ... file-n
   ```

   Each of the `file-i` arguments is assumed to contain a list of white-space separated numbers. Your script should print out the sum of the products of all of the numbers within each file. For example, if `file-1` contains the numbers 2 and 3, and `file-2` contains the numbers 4 and 5, then your output should be `26` (2 times 3 plus 4 times 5). If no files are given on the command line, the output should be 0. If a file contains no numbers (i.e. is empty, except possibly for whitespace) its product should be 1.

   There is a shell variable that you can use in your script to get a list of all command-line arguments. It can be found on your Linux reference sheet. You may assume that the files given do not contain spaces or other characters in their names that would make processing them difficult.

Testing tools

Note: the scripts you write in the following questions will be useful every time you write a program. Be sure to complete them! In this course, you will be responsible for your own testing. As you fix bugs and refine your code, you will very often need to rerun old tests, to check that existing bugs have been fixed, and to ensure that no new bugs have been introduced. This task is *greatly* simplified if you take the time to create a formal test suite, and build tools to automate your testing. In the following questions, you will implement such tools as `bash` scripts.

4. Create a `bash` script called `runSuite` that is invoked as follows:

   ```
   ./runSuite suite-file programToTest sampleProgram
   ```

   The argument `suite-file` is the name of a file containing a list of filename stems (more details below), the argument `programToTest` is the name of the program that is meant to be tested, and the argument `sampleProgram` is the name of the program whose behaviour `programToTest` is meant to match.

   In summary, the `runSuite` script runs `programToTest` and `sampleProgram` on each test in the test suite (as specified by `suite-file`) and reports on any tests in which the output of the two programs does not match.

   The file `suite-file` contains a list of stems, from which we construct the names of files containing the command-line arguments of each test. Stems will not contain spaces. For example, suppose our suite file is called `suite.txt` and contains the following entries:

   ```
   test1 test2
   reallyBigTest
   ```

   Then our test suite consists of three tests. The first one (`test1`) will use the file `test1.args` to hold its command-line arguments, the second one (`test2`) will use the file `test2.args` to hold its command-line arguments, and the last one (`reallyBigTest`) will use the file `reallyBigTest.args` to hold its command-line arguments.

   A sample run of `runSuite` would be as follows:

   ```
   ./runSuite suite.txt ./myprogram ./sampleProgram
   ```

The script will then run `./myprogram` and `sampleProgram` three times, once for each test specified in `suite.txt`:

- The first time, it will run `./myprogram` and `sampleProgram` with command-line arguments provided to the program from `test1.args`, and compare the results of of the two programs.
- The second time, it will run `./myprogram` and `sampleProgram` with command-line arguments provided to the program from `test2.args`, and compare the results of of the two programs.
- The third time, it will run `./myprogram` and `sampleProgram` with command-line arguments provided to the program from `reallyBigTest.args`, and compare the results of of the two programs.

Note that if the test suite contains a stem but a corresponding `.args` file is not present, the program is run without providing any command-line arguments.

If the output of `programToTest` for a given test case differs from the output of `sampleProgram` for that test case, print the following to standard output (assuming test `test2` failed):

```
Test failed: test2
Args:
(contents of test2.args, if it exists)
Expected:
(contents of the output of running the sampleProgram for the given test)
Actual:
(contents of the output of running the programToTest for the given test)
```

with the `(contents ...)` lines replaced with actual result of running the programs, without any changes, as described. The literal output `Args:` must appear, even if the corresponding file does not exist. Note that there is no whitespace after the colon for each of `Args:`, `Expected:`, and `Actual:` except for the newline character.

Follow these output specifications *very carefully.* You will lose a lot of marks if your output does not match them. If you need to create temporary files, create them in `/tmp`, and use the `mktemp` command to prevent name duplications. **Also be sure to delete any temporary files you create in `/tmp`.**

You can find sample solution of `runSuite` named `q4Example` in the directory `a1/q4` on your Git repository. Using the contents of this directory, running your script as:

```
./runSuite test_suite.txt ./my_factorial_buggy ./my_factorial_correct
```

should produce an output identical to the example provided to you by running the command

```
./q4Example test_suite.txt ./my_factorial_buggy ./my_factorial_correct
```

Note: Do **NOT** attempt to compare outputs by storing them in shell variables, and then comparing the shell variables. This is a very bad idea, and it does not scale well to programs that produce large outputs. We reserve the right to deduct marks (on this and all assignments) for bad solutions like this would be.

You can get most of the marks for this question by fulfilling the above requirements. For full marks, your script must also check for the following error conditions:

- incorrect number of command-line arguments to `runSuite`

- unreadble `.args` files for a test case, i.e. if the stem `myTest1` exists and the file `myTest1.args` exists but is not readable (it must exist as the behaviour if it doesn't exist is just to execute with no command line arguments).

If such an error condition arises, print an informative error message to standard error and abort the script with a **non-zero** exit status. Your program will only be evaluated on whether or not it produces a non-zero status code, however as you will be using this script to test your programs in the coming assigments it is in your best interest that the error messages be informative.

5. In this question, you will generalize the `runSuite` script that you created in problem 4. As it is currently written, this script cannot be used with programs that take input from standard input. For this problem, you will enhance `runSuite` so that, in addition to (optionally) passing command-line arguments to the program being executed, the program can also be (optionally) provided input from standard input. The interface to the scripts remains the same:

```
./runSuite suite.txt ./programToTest ./sampleProgram
```

The format of the suite file remains the same. But now, for each `testname` in the suite file, there might be an optional `testname.in`. If the file `testname.in` is present, then the script `runSuite`) will run `programToTest` and `sampleProgram` with the contents of `testname.args` passed on the command-line as before and the contents of `testname.in` used for input on `stdin`. If `testname.in` is not present, then the behaviour is almost identical to problem 4 (see below for a difference in the output): `myprogram` and `sampleProgram` are run with command-line arguments coming from `testname.args` with nothing supplied as input on standard input.

The output of `runSuite` is changed to now also show the input provided to a test if the test failed. Assuming test `test2` from Q4 failed, the output generated by the updated `runSuite` is as follows:

```
Test failed: test2
Args:
(contents of test2.args, if it exists)
Input:
(contents of test2.in, if it exists)
Expected:
(contents of the output of running the sampleProgram for the given test)
Actual:
(contents of the output of running the programToTest for the given test)
```

with the `(contents ...)` lines replaced with actual result of running the programs, without any changes, as described. The literal output `Args:` and `Input:` must appear, even if the corresponding files do not exist. Note that there is no whitespace after the colon for each of `Args:`, `Input:`, `Expected:`, and `Actual:` except for the newline character.

You can find sample solution of `runSuite` named `q5Example` in the directory `a1/q5` on your Git repository. Using the contents of this directory, running your script as:

```
./runSuite test_suite.txt ./my_factorial_buggy ./my_factorial_correct
```

should produce an output identical to the example provided to you by running the command

```
./q5Example test_suite.txt ./my_factorial_buggy ./my_factorial_correct
```

All error-checking that was required in problem 4 is required here as well.

(a) Modify `runSuite` to handle input from standard input.

(b) unreadble `.in` files for a test case, i.e. if the stem `myTest1` exists and the file `myTest1.in` exists but is not readable (it must exist as the behaviour if it doesn't exist is just to execute without redirecting input).

Note: To get this working should require only very small changes to your solution to problem 4.

Note: While your `runSuite` will run a program with no input if the `.in` file does not exist, if the program you are testing still expects input then your runSuite will hang waiting for input (as it is executing the program that expects input). Thus, for testing programs that require input if you want to test them with no input you should use an empty `.in` file rather than no file at all, so that they still receive EOF.

Submission:
The following files are due at Due Date 1: `a1q1a.txt`, `a1q1b.txt`, `a1q1c.txt`, `a1q1d.txt`, `a1q1e.txt`, `a1q1f.txt`, `a1q1g.txt`, `a1q1h.txt`, `a1q1i.txt`, `a1q1j.txt`, `a1q2a.txt`, `a1q2b.txt`, `a1q2c.txt`, `a1q2d.txt`, `a1q2e.txt`.

The following files are due at Due Date 2: `sumOfProducts` `runSuite`, `runSuite`.