

# CS246—Assignment 1 (Winter 2015)

N. Naeem and K. Bradley

Due Date 1: Monday, Jan 19, 04:55pm

Due Date 2: Friday, Jan 23, 04:55pm

**Questions 1 and 2 are due on Due Date 1; the remainder of the assignment is due on Due Date 2.**

1. Provide a Unix command line to accomplish each of the following tasks. Your answer in each subquestion should consist of a single pipeline of commands, with no separating semicolons (;). Before beginning this question, familiarize yourself with the Unix commands outlined on the Unix handout. Keep in mind that some commands have options not listed on the sheet, so you may need to examine some man pages. With the exception of `awk` in part (g), every command you need is on the Unix handout. Do not attempt to solve these tasks with `find`.

- (a) Print the number of *words* in `/usr/share/dict/words`.  
Place your command pipeline in the file `a1q1a.txt`.
- (b) Print the (non-hidden) contents of the current directory in reverse chronological order.  
Place your command pipeline in the file `a1q1b.txt`.
- (c) Print the first 10 lines from the text file `myfile.cc` that start with `#include` or `#define`.  
Place your command pipeline in the file `a1q1c.txt`.
- (d) Print up to the last 5 lines that contain the string `cs246` 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 uppercase or lowercase.  
Place your command pipeline in the file `a1q1e.txt`.
- (f) Print all (non-hidden) files in any *subdirectory* of the current directory that end with `.c` (immediate subdirectories only, not subdirectories of subdirectories).  
Place your command pipeline in the file `a1q1f.txt`.
- (g) 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 Unix 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 `a1q1g.txt`.

- (h) Out of the first 20 lines of `myfile.txt`, how many contain at least two digits? Place the command pipeline that prints this number in the file `a1q1h.txt`.
- (i) Print the 80<sup>th</sup> to 100<sup>th</sup> (both inclusive) lines in the file `myfile.txt`. Place the command pipeline in the file `a1q1i.txt`.
- (j) Print a listing, in long form, of all non-hidden entries (files, directories, etc.) in the current directory that are writable by at least one of owner, group, other (the other permission bits could be anything). 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, on a single line. If your pattern contains special characters, enclose it in quotes.

(a) Lines that contain cs246.

Place your answer in the file **a1q2a.txt**.

(b) Lines that contain cs246 or cs247 (or both).

Place your answer in the file **a1q2b.txt**.

(c) Lines that contain an occurrence of **<title>**, followed eventually by an occurrence of **</title>**.

Place your answer in the file **a1q2c.txt**.

(d) Lines that contain nothing but a single string of zombie sound effects (BRAINS, GRR, UGH, and RAWR) where such a string is of the form

**GRR!UGH!UGH!RAWR!BRAINS!GRR!**, with arbitrarily many zombie sound effects separated by an exclamation point (!). Place your answer in the file **a1q2d.txt**.

(e) Lines that contain nothing but a single string of generalized zombie sound effects, which are like ordinary sound effects but can have arbitrary numbers of the same vowel(s) and arbitrary number of Rs for GRR. For example,

**BRAAAIIINS!GRRRR!GRRRRRR!UUUGH!RAAAAAAWR!**. Place your answer in the file **a1q2e.txt**.

(f) Lines whose every odd character is a digit.

Place your answer in the file **a1q2f.txt**.

(g) Lines that contain nothing but a single occurrence of a string with the format **l1,l2A l3,l4** where **l1**, **l2**, **l3** and **l4** are line numbers and **A** can be one of the characters **a**, **d** or **c**.

Place your answer in the file **a1q2g.txt**.

(h) Lines consisting of a declaration of a single C variable of type **int**, without initialization, optionally preceded by **unsigned**, and optionally followed by a single line **// comment**. Example:

```
int varname; // 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). Place your answer in the file **a1q2h.txt**.

3. Write a Bash script called **dup** that takes one argument on the command line and prints it twice. For example:

```
$ ./dup hello
hello hello
```

You may assume that the user will call this script correctly; no error checking is needed.

4. Write a Bash script called **DiffThree** that takes three file names as command line arguments. **./DiffThree file1 file2 file3** should print the string

**The same.**

if the three files have the same contents. Otherwise, it should print the string

**Different.**

You may assume that the user will call this script correctly; no error checking is needed.

5. **Note: the script you write in this question will be useful every time you write a program. Be sure to complete it!** 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 a tool to automate your testing. In this question, you will implement such a tool as a Bash script.

Create a Bash script called `runSuite` that is invoked as follows:

```
./runSuite suite-file program
```

The argument `suite-file` is the name of a file containing a list of filename stems (more details below), and the argument `program` is the name of the program to be run.

In summary, the `runSuite` script runs `program` on each test in the test suite (as specified by `suite-file`) and reports on any tests whose output does not match the expected output.

The file `suite-file` contains a list of stems, from which we construct the names of files containing the input and expected output of each test. 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.in` to hold its input, and `test1.out` to store its expected output. The second one (`test2`) will use the file `test2.in` to hold its input, and `test2.out` to store its expected output. The last one (`reallyBigTest`) will use the file `reallyBigTest.in` to hold its input, and `reallyBigTest.out` to store its expected output.

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

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

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

- The first time, it will run `./myprogram` with standard input redirected to come from `test1.in`. The results, captured from standard output, will be compared with `test1.out`.
- The second time, it will run `./myprogram` with standard input redirected to come from `test2.in`. The results, captured from standard output, will be compared with `test2.out`.
- The third time, it will run `./myprogram` with standard input redirected to come from `reallyBigTest.in`. The results, captured from standard output, will be compared with `reallyBigTest.out`.

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

```
Test failed: test2
Input:
(contents of test2.in)
Expected:
(contents of test2.out)
Actual:
(contents of the actual program output)
```

with the (contents ...) lines replaced with actual file contents, as described. **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 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
- missing or unreadable `.in` or `.out` files (for example, the suite file contains an entry `xxx`, but either `xxx.in` or `xxx.out` doesn't exist or is unreadable).

If such an error condition arises, print an informative error message to standard error and abort the script with a nonzero exit status.

6. **Note: the script you write in this question will be useful every time you write a program. Be sure to complete it!** In this question, you will start with the `runSuite` script that you created in problem 5, and generalize it. As it is currently written, `runSuite` only works for programs that take their input on `stdin`; it cannot be used with programs that take parameters on the command line. For this problem, you will enhance `runSuite` so that it can pass command line arguments. The interface to `runSuite` remains the same:

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

The format of the suite file remains the same. But now, for each `testname` in the suite file, there will be files `testname.in`, `testname.out`, and an optional third file `testname.args`. If the file `testname.args` is present, then `runSuite` will run `myprogram` with the contents of `testname.args` passed on the command line and the contents of `testname.in` used for input on `stdin`. If `testname.args` is not present, then the behaviour is identical to problem 5: `myprogram` is run without arguments, and `testname.in` still supplies the input on `stdin`. The file `testname.out` is used identically to the way it was used in problem 5, and the output of `runSuite` should also be identical to the way it appeared in problem 5. All of the error-checking that was required in problem 5 is required here as well.

**Note:** To get this working should require only very small changes to your solution to problem 5.

### 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`, `a1q2f.txt`, `a1q2g.txt`, `a1q2h.txt`.

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