CS 251 Outlab 1: UNIX CLI and BASH

Interesting fact:
Often you'll find yourself wondering if you should add a '/' after a variable that you know is a directory name. For e.g., if a="dir", should it be ${a}/blah.txt? What if the input was "dir/"? Turns out, dir//blah.txt, and dir/blah.txt, both work! In fact any number of slashes work just fine!

Before you begin-

Understand the difference between input from STDIN versus input from the command line. This will haunt you now, and in the upcoming assignments. STDIN is what you typically use to provide input to a C++ executable by executing "./a.out", and then typing in input, or "piping" it in from a file (./a.out < file). Command line arguments are what you give to g++. For example, to compile c.cpp, you would probably execute "g++ c.cpp" which would produce a.out. Here 'c.cpp' is a "command line" argument/input.
Read carefully in each question if the input is via STDIN or command line. If it isn't clear do not hesitate to ask the TAs on forums.
Remember - there are no stupid questions :)

This Outlab is worth a total of 150 points. Do this in pairs.

The Resources-1 provided to you contains C++ codes to test in Task3, a folder of songs for Task4 and a folder of text files for Task5. Check this document for urls to test Task2.

Task 1 - Integers, your true shell friends

[Total 13 points, you can earn 5 bonus points!]
(variables, functions, process substitution, loops)

1.A sum two numbers **[1 point]**

Usage: ./sum.sh <num1> <num2>

This task is supposed to introduce you to the world of bash variables, particularly, integer values. Bash is not good at tracking floating points (not natively at least, you can find separate commands/packages!)
Your task is to primitively implement the command **expr**, without of course using it. Let's start off with integer addition.
Given two numbers as **command line arguments** that are numbers**,** calculate the sum of these two numbers. Print "Wrong number of arguments, expected 2", if fewer or more than 2 arguments are presented, and **exit** with status 1. Else print the sum exit with status 0.

1.B sum of all numbers **[2 points]**

Usage: ./sum_all.sh <num1> …

Now instead of two numbers, you're given an arbitrary number of arguments

Given two numbers as **command line arguments** that are numbers**,** calculate the sum of these two numbers. Print “No numbers given”, if fewer than 1 argument is presented, and **exit** with status 1. Else print the sum and exit with status 0. You can use your script from 1.A here, in fact it is advisable to do so.

1.C Convert to Roman Numerals **[10 points]**

Usage: ./roman.sh <number>

Given 1 argument, convert it into roman numeral. The range of this number is 1 to 200 (can’t be zero, Romans forgot about the nought). Assume exactly one argument is given and it’s a decimal integer between 1 to 200 (don’t sanitise). Find out the rules for making a roman numeral, and try to reduce the code size as much as possible. Your code will be objectively evaluated based on the correctness, of course, and the number of characters in your code (excluding those lines that start with a ‘#’, so make sure comments are in a new line). You are not allowed to submit any other file along with this script to reduce your script size. **This script and only this script** will be copied to an empty folder to be tested. All code should be in this script, and no other helper (downloaded from the internet) is allowed to be used, there will be severe penalty for any petty tricks

<= 400 characters - 15 points (**5 bonus!**)

(Hint: One way to do it involves process substitution and functions)

<= 500 characters - 12 points (2 bonus!)

(Would still require to eliminate some repetition)

<= 1000 characters - 10 points

More than 1000 characters - 5 points

How to check the number of characters in your file removing new line comments:

```
sed '/^\(\s\)*#/d' roman.sh | wc -c
```

Should you wish to minimize code by removing whitespace, I am providing you the character limits for all non-space characters in your code

<= 350 characters - 15 points (**5 bonus!**)

(Hint: One way to do it involves process substitution and functions)

<= 450 characters - 12 points (2 bonus!)

(Would still require to eliminate some repetition)

<= 900 characters - 10 points

More than 900 characters - 5 points

How to check the number of characters in your file removing new line comments and whitespace:

```
sed '/^\(\s\)*#/d' roman.sh | sed 's/\s//g' | wc -c
```

Task 2 - God save the Queen (based on a true historical event)

(tail, tr, grep)

[Total 37 points]

Usage: ./cipher.sh <url>

(Print the above line if the user doesn't enter the required number of arguments, and **exit** with status 1)

The url will be provided as an argument to your script.

It is 1586. Queen Elizabeth I reigns over England. You are Phelippes, a code decipherer who works for Sir Walsingham, the Queen's loyal spymaster. Sir Walsingham suspects that Mary, Queen of Scots is scheming against Queen Elizabeth, and has imprisoned her in Chartley Castle. You are posted there in order to unravel the plot.

Your job is to intercept Mary's correspondence and look for incriminating evidence. This isn't so straightforward- you know that Mary encrypts her letters!
Write a shell script **cipher.sh** to do the following:

1. Download the url that is provided as an argument and save it as encrypted.txt
   Testcase url: https://www.cse.iitb.ac.in/~vahanwala/CS251/babington/letter.txt

2. You know this is a letter that Mary has written! It is encrypted using a rather simple **Caesar shift cipher** eg. if A->G then B->H, C->I, …, Z->F (Breaking the fourth wall: Probably not historically accurate)

   You know that the last line of the letter has her signature block. Try all possible shifts for the last line or two of encrypted.txt , and check if the deciphered text contains "Mary" or "Queen" or "Majesty" (Hint: the tail, tr and grep commands should be useful. Your grep search should be case-insensitive)

3. Having figured out the key to the cipher, decipher the whole letter and save it in deciphered.txt If you've done it right, this should be English we can understand. Make sure that you preserve the formatting in terms of newlines. We expect the use of the tr command.

4. Indeed, Mary is guilty of high treason, and you must nab all the conspirators. It's time to use the broken cipher to your advantage! Using the key to the cipher, encrypt the line "PS. Give me the names." and append this encrypted line to encrypted.txt (the line is without quotes)

You have to submit cipher.sh, we will generate encrypted.txt and deciphered.txt when we run your script.

You will get 25 points if your deciphered.txt matches, but the final encrypted.txt doesn't

Task 3 - Check your code!

[Total 40 points]

(wget, basename, g++, diff, wc)

Usage: ./verifier.sh <source file> <testcases url> <cut-dirs arg>

(Print the above line if the user doesn't enter the required number of arguments, and **exit** with status 1)

Example: ./verifier.sh ~/Downloads/failure.cpp
https://www.cse.iitb.ac.in/~vahanwala/CS251/verifier/ 2
This is a legit url that works.

The third argument is to help you save the contents in an appropriately named directory. In the above example, your download will be saved in a directory called "verifier".

Another example:
https://host.name.domain/~username/CS251/Outlab1/example/ 3 will save the contents in a directory called "example". This will be a subdirectory of the working directory.

After the wget command, your working directory should have the following structure:

```
.
├── verifier.sh
└── <basename of the url directory>/
    ├── inputs/
    │   ├──<name1>.in
    │   ├──<name2>.in
    │   ├──<name3>.in
    │   ├── … (several other input testcases)
    │   └──<some name>.<some other extension>
    └── outputs/
        ├──<name1>.out
        ├──<name2>.out
        ├──<name3>.out
```

```
        ├── … (corresponding true outputs for each input)
        └──<some name>.<some other extension>
```

This is the same structure as the online directory.
The files are named helpfully: the correct output to **inputs/foo.in** is given in **outputs/foo.out**

Now, the first argument tells you where the C++ source code is. Use the script to copy it to the Task3 directory and compile it. You would like to see if the code passes all testcases!

Run the executable against all the inputs: these are the files in the **inputs** directory with the ".in" extension. Save the (suitably named) outputs in a directory called **my_outputs**. This will help you in comparing the output of the given C++ code with the expected output.

Prepare a file called **feedback.txt**: this will tell you what went wrong. The first line should be "Failed testcases are:" (*without quotes*)
In the following lines, write the basenames of the testcases that failed, one on each line. For example, if **my_outputs/foo.out** and **outputs/foo.out** do not match, your entry in feedback.txt should be simply "foo" (without quotes)

Finally, use the script to check if all testcases passed. If they did, print a single line on the terminal: **"All testcases passed!"** (without quotes)

If not, print **"Some testcases failed."** on the terminal (without quotes). At the end of all this, here is what your directory structure should look like:

```
.
├── verifier.sh
├── feedback.txt
├── <name of source file>.cpp
├── a.out
└── <basename of the url directory>/
    ├── inputs/
    │   ├── <name1>.in
    │   ├── … (several other input testcases)
    │   └── <some name>.<some other extension>
    ├── outputs/
    │   ├──<name1>.out
    │   ├── … (corresponding true outputs for each input)
    │   └── <some name>.<some other extension>
```

```
└── my_outputs/
    ├── <name1>.out
    └── … (corresponding code outputs for each input)
```

Just wget done: 10 points
Directory structure matches after script, but wrong feedback and/or terminal output: 30 points

Task 4 - Music! (ln, loops, basename, mkdir)

[Total 30 points]
Usage: ./organiser.sh

You are given a directory **songs/** which, well, contains a list of files, each named after a song. *For this task, just submit the shell script, assume that the **songs/** directory will be present in the working directory when your script is run.*

What will be the contents of the files? The first line will indicate which album it is from. The second line will be an integer *N*- this is how many playlists the song features in. The last *N* lines list the playlists the song is a part of. *N* can be 0.

For example, consider

```
$cat Heathens
Suicide_Squad
3
favourites
workout
haunting
$
```

Your task is to - you guessed it - organise these songs. Create two more directories - **albums/** and **playlists/**

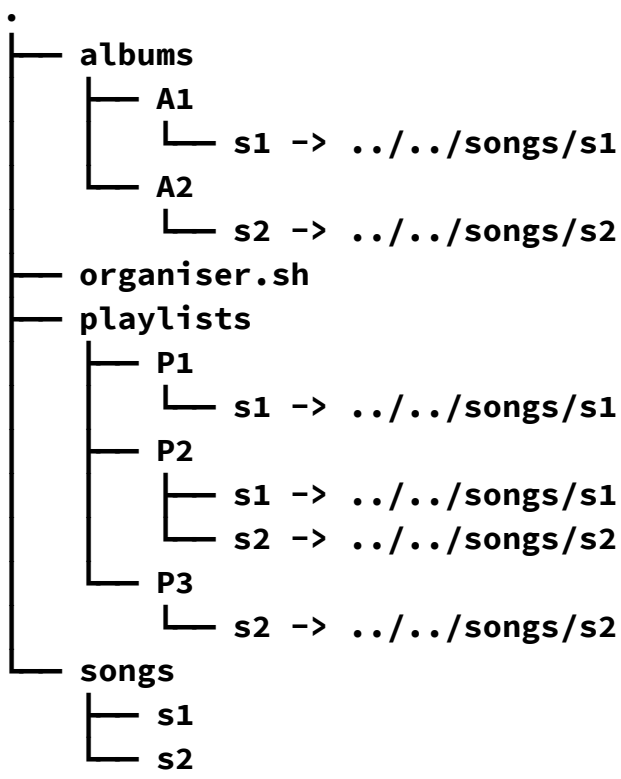
Now, to save space, you obviously don't want to spawn copies of the song all across the directory. Your script should make a folder for each album and playlist. These folders should contain symbolic links to the songs associated with them. The symbolic links should have the same name as the song.

Suppose the **songs/** directory has two songs, s1 and s2

```
$cat songs/s1
A1
```

```
2
P1
P2
$cat songs/s2
A2
2
P2
P3
```

After your script is done, here is what your directory structure should look like



You will get 10 points if you organise the songs into albums but don’t create playlists. Or vice versa.

Task 5 - The Hidden Lore

[Total 30 points]

(grep, cut, exit status, loops)

Usage: ./story.sh <directory> <first key> <output-file>

You're given a directory full of text files. There are exactly 3 lines of text in each file, and each line is non-empty (has words). The first lines of all these files together tell a story or sing a song, however the files are in no conceivable order. The second line of each file has a single word, which from here on we'll call the **key** to this file, and the third line of each file too has a single word, which will be the **key** to the next file. You're given the directory containing these files, a starting key as command line argument, and an output file. Your task is to find the file using this key, output its first line to the output file, grab the next file's key from this file, and **recurse,** until there is no file with the current key. **Remember that the key to a file is always in it's second line, and nowhere else. Even if the word matches any other line, that is not the file's key. We'll make sure the second and third line are always distinct, and there are no cycles in the traversal, to avoid an infinite loop.**

(for convenience, there will be no ":" in the words or the filenames. Think about why)

Submission Instructions

The following is what your submission directory must look like. Please be considerate and follow the naming conventions strictly, because grading is automated. To check, you can apt-get tree and run the tree command.

```
.
|-- Task1-Integers
|   |-- roman.sh
|   |-- sum.sh
|   `-- sum_all.sh
|-- Task2-GodSave
|   `-- cipher.sh
|-- Task3-CheckCode
|   `-- verifier.sh
|-- Task4-Music
|   `-- organiser.sh
`-- Task5-HiddenLore
    `-- story.sh
```

Instructions on what to name your submission directory and where and how to submit will be updated later this week, so please stay tuned. For now, you can start thinking about how you're going to solve these problems.
The deadline will be sometime next weekend.

You must take responsibility for every line of code you write.

Please don't plagiarize. It usually isn't pretty for those who do.