Outlab 1: Shell scripting and git (100 points)

This Outlab is to be done in pairs, unless we've informed you that you're an exception. Pairing info here. Make sure you read the instructions at the end before submitting.

Task 1: BASH. Make An Autograder! (55 points)

For those of you who want a pointer to get started with bash scripting, this is nice.

In this task, we will familiarise ourselves with the UNIX command line, and explore how scripts can help us automate some useful tasks. *Your code for this task will also be evaluated by a script, so please be careful and follow the naming conventions we specify.*

Automated grading is something you'll see more than once, and hopefully, doing this assignment will really *show* you why everyone insists that you be disciplined while naming your files.

So here's the scene. You're a TA for your favourite lab course. There has been an assignment. You've been assigned the task of verifying that the code people have turned in indeed gives the correct results, and awarding credit accordingly. You have to write three shell scripts for this task: `download.sh`, `organise.sh` and `evaluate.sh`. These scripts will be under the same directory.

As for awarding *you* credit for your scripting, **marks for each of these parts will be awarded independently**. That is, you can assume that while we grade your work, your `organise.sh` will see the correct state from `download.sh`, and so on. Still, you will get the most satisfaction from this assignment if you can get everything right.

We take a small example to explain the directory structure at each step (you can use the `tree` command to verify). We start with:

```
.
|-- download.sh
|-- evaluate.sh
`-- organise.sh
```

Part 1: download.sh (wget, basename) (10 points)

To start, you've been given a link (like https://www.cse.iitb.ac.in/~vahanwala/ssl21/mock_grading/; when you write this link, don't forget Slash at the end, or you'll just end up walking in the cold November Rain) from where you can pull the documents you need. By the way, https://www.cse.iitb.ac.in/~vahanwala/ssl21/ also has another testcase (https://www.cse.iitb.ac.in/~vahanwala/ssl21/big_mock_case/) click the link to check. You need to write a script, `download.sh`, that is invoked as follows:

```
# bash download.sh <link to directory> <cut-dirs argument>
```

First notice that this script needs two command line arguments to work as intended. Thus, it should first check that it has been supplied exactly two command line arguments before proceeding further. If the check fails, it should print the following line (without quotes):

"Usage: bash download.sh <link to directory> <cut-dirs argument>"
and exit with code 1.

On running as intended, this should save the online directory locally as `mock_grading`. This directory will be under the same directory as that of the scripts. The local directory should have the *same structure* as the online one, the only difference being that the local version should not have the `index.html` files. `wget` does have an option of "rejecting" certain files. You may still have unwanted files, with the `.tmp` extension. Try pattern matching along with the aforementioned option.
Use `wget` in the quiet mode for this task. It is clear that you want to use the recursive option, and you do not want to download stuff from parent directories. If you don't specify anything further, you'll see that the downloaded directory you actually want is nested in directories, starting from the one corresponding to the host, eg. `www.cse.iitb.ac.in`, within that `~vahanwala`, within that `ssl21` and finally within *that* is the directory you actually need.
`wget` has an option for not creating a directory corresponding to the host, but even if you invoke that, you're still stuck with 2 levels of nesting that you don't really want. That is where the `cut-dirs` flag will help. You don't need to worry about your script being fed the wrong argument for this.
Finally, your script needs to make sure that the local directory is called `mock_grading`. The online directory may have a different name. Check if that's the case (`basename` might do the trick), and if so, rename the directory you just got locally.
At this point, the directory structure should look something like (assume that `inputs`, `outputs`, `submissions` do not have any further subdirectories):

```
.
|-- download.sh
|-- evaluate.sh
|-- mock_grading
|   |-- inputs
|   |   |-- 0.in
|   |   `-- 1.in
|   |-- outputs
|   |   |-- 0.out
|   |   `-- 1.out
|   |-- roll_list
|   `-- submissions
|       |-- 180070026bar.cpp
|       |-- 180070035.cpp
|       `-- 180070035foo.txt
`-- organise.sh
```

Note: The cut-dirs parameter will be specified in such a way that just the wget command will save the downloaded directory locally with the same name as the remote one. Eg. in this example, it will be 2.

Part 2: organise.sh (ln, for) (20 points)

Alright! You now have the roll list of students, the stuff they have turned in, the inputs to test their code, and the expected output corresponding to each input.

There's a small problem though. If you look in the `submissions` directory, you'll find that everyone's files are together. If you run code from here, it might just so happen that Bob's `.cpp` file finds Alice's `.h` file particularly helpful. Thankfully, the files are named honestly: every file begins with the roll number of the student who submitted it, and we assume that all roll numbers are of equal length.

Our `organise.sh` will be called as follows:

```
# bash organise.sh
```

On running, it should create a directory called `organised` under the same directory as the scripts. Within the `organised` directory, it should create directories named after every roll number, eg. `180070026/, 180070035/, …` (see `roll_list` under the `mock_grading` directory for the complete list of roll numbers)

You might have guessed where this is going. However, we are not going to copy paste files from `mock_grading/submissions` into these roll number directories. Some of these files may be large, and this might waste space. What we'll do instead is simply create symbolic links (`ln -s`) to each of the files that the student has submitted.

For example, if roll number 180070035 has submitted `180070035.cpp`, then in `organised/180070035`, the file `180070035.cpp` is a symbolic link that'll point to `../../mock_grading/submissions/180070035.cpp`

Make sure you use relative paths while creating the symbolic links.

Now, the directory structure should look like:

```
.
|-- download.sh
|-- evaluate.sh
|-- mock_grading
|   |-- inputs
|   |   |-- 0.in
|   |   `-- 1.in
|   |-- outputs
|   |   |-- 0.out
|   |   `-- 1.out
|   |-- roll_list
|   `-- submissions
|       |-- 180070026bar.cpp
|       |-- 180070035.cpp
|       `-- 180070035foo.txt
|-- organise.sh
`-- organised
    |-- 180070026
    |   `-- 180070026bar.cpp -> ../../mock_grading/submissions/180070026bar.cpp
    `-- 180070035
        |-- 180070035.cpp -> ../../mock_grading/submissions/180070035.cpp
        `-- 180070035foo.txt -> ../../mock_grading/submissions/180070035foo.txt
```

For simplicity, you can assume that everyone who is on the roll-list has submitted at least one file and at most one cpp file.

Part 3: evaluate.sh (g++, diff, timeout, redirection, sort) (25 points)

We are now ready to do the actual grading. You can assume that each student has submitted at most one `.cpp` file; these are the only files we care about in this step. We will run this script as

```
# bash evaluate.sh
```

On running, this script will generate two files: `marksheet.csv` and `distribution.txt`. These, as usual, will be in the same directory as our three scripts. Each of these files will have as many lines as there are students, i.e. roll numbers in `roll_list`.

`marksheet.csv` will have the score of students, sorted in lexicographical order of roll numbers, in the format:

<roll number>,<score>

on each line. For example:

180070035,2

`distribution.txt` will have just the scores, one on each line, sorted in descending order of the integer score.

How are these scores calculated though?

You need to do this for each student, from `organised/<roll no>`. First, compile the single `.cpp` source file into an executable called, wait for it, `executable`. The file may not exist, or may throw some compilation error, which causes `g++` to complain. You're a busy TA, you don't have time to deal with errors that aren't your own.

Suppress the error messages with `2>/dev/null` (`2` refers to `stderr` which is, as it suggests, the standard file descriptor to which error messages are written, and what we do here is suppress stuff by redirecting it to a null device)

Now, the `inputs` and `outputs` in `mock_grading` come into play. For each input `mock_grading/inputs/foo.in`, the expected output is in `mock_grading/outputs/foo.out`

You need to run `executable` against each of the inputs, one by one, and store the generated output for `mock_grading/inputs/foo.in` in `organised/<roll no>/student_outputs/foo.out`

You'll have to do something like

```
# ./executable < input > output
```

The nice thing here is that the `input` file will of course exist, and the `output` file will be generated regardless of whether `executable` actually exists. Try it yourself!

But be careful! This is student code, there can be errors like segmentation faults, so you also need to redirect error messages to `/dev/null` just like you did while compiling. There's one more catch: you can't let code run forever, so set a timeout of 5 seconds for running each testcase. For convenience, please also suppress Segfault error messages by piping to the null command.

For each testcase, compare the generated output with the expected output. If they are **exactly** the same, add 1 to the student's score, else, do nothing. **Don't worry, we will only test you on cases where the expected output is not blank.**

Thus, you populate `marksheet.csv` as you evaluate each test case for each student. From here, making `distribution.txt` won't be too hard.

Here's what your working directory will look like, finally: (in this instance, it so happened that `180070026bar.cpp` threw a compiler error) **From the diagram, it should be clear that you're doing the compilation and testing from within `organised/<roll_no>`**

Disclaimer: Suppressing error messages is just a hack that is *sometimes* handy. For other applications, it is usually a bad idea.

```
.
|-- distribution.txt
|-- download.sh
|-- evaluate.sh
|-- marksheet.csv
|-- mock_grading
|   |-- inputs
|   |   |-- 0.in
|   |   `-- 1.in
|   |-- outputs
|   |   |-- 0.out
|   |   `-- 1.out
|   |-- roll_list
|   `-- submissions
|       |-- 180070026bar.cpp
|       |-- 180070035.cpp
|       `-- 180070035foo.txt
|-- organise.sh
`-- organised
    |-- 180070026
    |   |-- 180070026bar.cpp -> ../../mock_grading/submissions/180070026bar.cpp
    |   `-- student_outputs
    |       |-- 0.out
    |       `-- 1.out
    `-- 180070035
        |-- 180070035.cpp -> ../../mock_grading/submissions/180070035.cpp
        |-- 180070035foo.txt -> ../../mock_grading/submissions/180070035foo.txt
        |-- executable
        `-- student_outputs
            |-- 0.out
            `-- 1.out
```

Task 2: Git (45 points)

In this task we will get you acquainted with git.

Git is a VCS (*Version Control System)* that lets you easily manage several versions of your code and collaborate with your teammates, no matter how small or large your code base is.

You can manage your work locally in a *Git repository*, or *"push"* your work to a remote server such as the widely used *GitHub* or our own IITB CSE GitLab. Moreover, you can connect your local repo with the remote repo to sync the changes. Connecting to the same

remote repo allows your teammates to see all the changes that you *"push"* to the *remote* repo. If granted permission, your teammates too can push their changes to the code to your *remote* repo, which basically allows you to collaborate with your teammates.

Although you will have to submit most of your assignments through Moodle, we suggest that you start using git right away to collaborate with your teammates, it will make your life much easier later.

Let's get you started!

Sub Task 1: (15 points)

(checkout, add, commit, push, pull, branch, merge, rebase)

Note: The person with the lexicographically smaller roll number is referred to as 'you' and the other one as 'your friend'

You and your friend are working as freelancers and your client requires you to implement the backend for their app's login page.

Part 1: First Commit (You) (2 points)

1. Create a ***private*** repository on IITb CSE Git Server from 'your' account. Name the repository as **<first-rollno>-<second-rollno>-git (if this name is not available add random extra digits at the end: <first-rollno>-<second-rollno>-git42)**

   Update: In case any member of your team does not have a CSE Ldap, you can create a **private** repo on GitHub. ~~Make sure the github accounts are created from your IITB Ldaps, i.e. your iitb email ending with @iitb.ac.in~~
   Use your regular email account for github. Make sure it's reasonably clear who you are from your id. (updated in light of recent email from head CC)

   If, for whatever reason, you can't use CSE git (no CSE LDAP, login doing weird stuff, fill this form and proceed with GitHub as described above)

2. Add 'your friend' as a collaborator

The following steps are to be performed by the 'you': (git clone, git add, git commit, git push)

1. Download ***passwords.h*** from moodle. Create a file called ***utils.h***. Implement a function with the following signature in it (don't forget to include passwords.h in it):

   ```
   bool login(string name, string password)
   ```

   This function returns true only if a user with the given name is present in the database(which is defined in ***passwords.h***) and the password matches. Use auxiliary functions defined in ***passwords.h*** to complete this function.

2. Create a ***main.cpp*** that takes in two strings from ***stdin***, a ***name*** and a ***password***. Call the function ***login*** from main. If it returns true, print ***Success!*** else print ***Login Failed :(***

3. Now add all the changes in your working directory to the ***staging*** area (git add) and commit these changes with the commit message ***feat: Login API***

Learning:

This will create a ***master branch*** *and your first* ***commit*** in git. A commit is like a snapshot of your work. If you make any changes to your working directory or create any new commits, you can always come back to any previously created commit and ***checkout*** what you were up to when you made that commit.

At its heart git is just a *content-addressable filesystem,* i.e. it stores files for you and you can retrieve any file, anytime, based on the content of that file. For doing that git internally calculates a ***hash*** for your file, and you can retrieve that file later by remembering the hash that git gives you. However, we don't do this dirty work ourselves. Git provides us with a nice set of *user-friendly* commands that hide away these details and make our lives easier.

Now, use the command ***`git log`*** to see the *history* of commits that you have made. You will only see a single commit, showing a string of 40 characters, saying that your current HEAD is at *master*, as follows:

```
commit 0e9513a09611d3d54197a458c2322061bcb1551f (HEAD -> master)
Author: guitarhero22 <rushabhkanadia@gmail.com>
Date:   Thu Jul 29 16:16:00 2021 +0530

    feat: Login API
```

Git stores everything and anything that you throw at it(even your commits) in its *content-addressable filesystem*. Using this 40 character *hash value*, you can refer to any commit you have made. However, it's a pain to remember these long hash values when your project is very big. For this, git provides us with objects called ***refs***.

Basically, git lets you attach names to some important commits / milestones in your project. So, to return/refer to a previous commit you can simply use this ***name***. Several types of ***refs*** are available in git, the most frequently used are: ***tags*** and ***branches***, and they behave differently.

Branch: Apart from other metadata, like the author, time, etc. for a commit, git also stores pointers to the commits that immediately came before it. A branch in a git is a movable pointer. As you make more commits on the current branch, this pointer moves forward and points to the latest commit.

> **Tag:** a tag, as opposed to a ***branch***, is a static pointer, mostly used as a milestone to refer to important updates or mark different versions of your code.
>
> All the files, log and git objects are stored inside ***.git*** folder inside the repo.

4. Create a file ***README.md***. And report the ***hash*** of the last commit that you made in it. Add this file to your staging area and create a new commit with the message **Adding README**.

5. Push your commits to the remote branch ***master*** (git push)

Part 2: Branch off (Your Friend) (3 points)

Your friend thinks it'd be really cool if you assigned user-ids to all the users. But you don't buy it and ask your friend for a Proof of Concept (POC). Your friend is too lazy to write all the code from scratch. So, they decide to ***branch off*** from your current master branch and develop their proof of concept on this branch.

Your friend needs to perform the following steps:

1. Clone the repository, if already cloned before the previous part, pull the latest commits (git pull)
2. Create a branch named ***thisisabetteridea***. And switch to this branch.
3. Download the file ***new_passwords.h*** from moodle and copy its contents to ***passwords.h*** in your current working directory.
4. You may or may not need to change the implementation of the function ***login*** in ***utils.h***.
5. Add changes to the staging area and commit your work with the commit message ***feat: The Better Idea***
6. Push your commit to the *remote* branch ***thisisabetteridea***

Part 3: Fixes (You) (2 points)

Your client is complaining that their users are confused and don't know what to do when they land on the login page. You realize that you forgot to prompt users asking for their names and passwords.

You need to do the following:

1. Change the ***main*** function in main.cpp, ask the user to ***Enter Name:*** and ***Enter Password:*** before reading the two strings from stdin.
2. Add your changes to the staging area and commit your work with the commit message ***Fix: Adding Login Prompts***
3. Push your commits to the remote branch master

Part 4: Rebase / Merge (Your Friend) (4 points)

Your friend finds out that you have made changes to the *master* branch after they *branched off*. Now, to test their code, your friend needs all the changes you made in their branch as well.

Your Friend can do one of the following things: (git rebase, git merge)

Rebase:

Rebase their branch (*thisisabetteridea)* to your *master* branch.

For this your friend needs to:

1. Checkout to their branch ***thisisabetteridea***
2. Use the appropriate git command to rebase to your *master* branch
3. Resolve conflicts if any and continue to rebase **if needed**
   (git rebase --continue)
4. Push their latest commits to the remote branch ***thisisabetteridea***

OR

Merge:

Merge your *master* branch into their branch ***thisisabetteridea***

For this your friend needs to:

1. Checkout their branch ***thisisabetteridea***
2. Use the appropriate git command to merge your *master* branch into ***thisisabetteridea***
3. Resolve Merge Conflicts if any and commit the changes with the appropriate commit message
4. Push their latest commits to the remote branch ***thisisabetteridea***

Part 5: (You) (4 points)

After your friend implements the POC, you finally agree to agree. Now, you need to merge all their work into your *master* branch which you are using to deploy the login page.

For this, you need to do the following things:

1. Checkout your master branch
2. Use the appropriate commands to merge his ***thisisabetteridea*** branch into your master branch.
3. Push your latest commits to the *remote master* branch

That's it, you completed the basic login page your client asked for. Onward to more challenging tasks.

Sub Task 2: Saving Private Creds (30 points)

This task can be done by you or your friend or both.

This task is small (4-5 different commands) but tricky.
For this task you will need to download **git-task2.zip** provided to you on moodle.

This time a client asked you to handle the backend for their website and take care of their server. The previous developers who worked on the project were careless and added sensitive server credentials somewhere way back in the git history. You cannot publish your repo with such sensitive data exposed. Anybody could hack into your client's backend.

You need to find out where (the commit) the server credentials (file ***creds.env***) were added to the repo and remove this file from the entire git history to publish the repo. You will fix the repo that we gave to you and submit it back to us.

Note that simply deleting that file from your master branch and creating a new commit won't be enough. Because then anyone can checkout one of your previous commits and still find out the server credentials.

Moreover, you cannot create a new repo by simply copying everything from *master* branch except ***creds.env*** because this repo is marked with important milestones (***tags***) and contains several versions of your server, which your client might need.

For completing this task you may have to create one or two new branches / tags or write commit messages. You are free to use any ***sensible*** names / messages you like.

Evaluation Scheme:

- If I can't find creds.env from the latest commit of the master branch, but I am able to retrieve it by checking out previous commits of master, with least damage done to the master branch history - 5 points
- If I can't retrieve creds.env from the master branch, with the least damage to rest of the history - 15 points
- If I can't retrieve ***creds.env*** from anywhere in the repo using any of the following commands: ***git log --reflog*** and ***git checkout*** from anywhere inside your rep, with the least damage done to the rest of the history - 30 points

(With git log/git show, trace the commit(call it commit X) that added ***creds.env***. Branch off from the previous commit. Restore all files from commit X in the current working directory. Add only the required files to the staging area and create a commit(could use .gitignore here). Rebase the master branch from the commit X onto the newly created commit. Use git reflog to remove dangling(unreachable) commits from reflog, then use 'git gc' to prune history . You might find the following commands useful - checkout, branch, restore, rebase --onto, git reflog expire, git gc)

Note: **git rebase** will not delete the commits from the previous branch, it creates new commits. So, one should still be able to retrieve creds.env, until and unless someone prunes the history, so that commits not reachable from any **refs** are deleted. Such dangling commits

can be deleted by using **git reflog expire** with appropriate flags to marks these **unreachable** commits as expired, and then pruning them with **git gc**

If you use the approach specified in the hint, when you use git log you will see only one commit with your username, rest all should be from the user **guitarhero22**

Do not create a new repo and artificially recreate the history. We will find out if you did so.

Notes:

- Hints are given in (this) form. If you find any commands in these hints **find out** how to use them and what they do, and then solve the problem
- This Guide has everything you need to know about Git. If you don't understand something that I've said in this assignment or you want to learn more, you can read up on that particular topic from here.

Submission Instructions:

Bash:

Your submissions for Bash will be **autograded**, so please adhere to the naming conventions strictly. For your submission, first make a directory called `<rollno1>-<rollno2>` where rollno1 is the lexicographically smaller roll number. Eg. `180070026-180070035`

The following should be the structure of your directory. Check with `tree -a`

```
.
|-- code
|   |-- download.sh
|   |-- evaluate.sh
|   `-- organise.sh
`-- references.txt
```

Compress this `<rollno1>-<rollno2>` directory, strictly with the command

`tar -czvf.` For example,

```
tar -czvf 180070026-180070035.tar.gz 180070026-180070035/
```

Git:

We will **manually** grade your submissions for Git.

Sub Task1:

Add users **rushabh** (for the time being) to your IITb CSE Git Server Repo.

Update:
IF you have a member who does not have a CSE Ldap, then create a private repo on github, and add the github user **`guitarhero22`** as a collaborator.

Sub Task2:

Create a file report.txt which contains the commands that you used to solve the problem. And explain why you used a particular command. Before submitting please check with **`ls -a`** that **`.git`** is present in **`git-task2`**

Create the submission directory as that looks like this:

```
<rollno1>-<rollno2>
|-- git-task2
`-- report.txt
```

Compress this directory with the following command:

~~`tar -czvf <rollno1>-<rollno2>-gt2.tar.gz <rollno1>-<rollno2>-gt2/`~~

```
tar -czvf <rollno1>-<rollno2>.tar.gz <rollno1>-<rollno2>/
```

There will be two submissions on Moodle, one each for bash and git. Submit from the account of the lexicographically smaller roll number. Consider Sunday August 8 11:59 pm as the deadline. We will outline the late submission policy shortly.

Late Submission Policy: Moodle will accept submissions for upto two hours after the deadline: i.e. exactly up to 1:59 am, Monday, August 9. We will not accept submissions after that. If you make your submission in this 2 hour window, you'll be penalised 1 mark out of 100. (a late submission for either bash or git, or both, results in 1 mark penalty)

However, if you truly can't make the submission on time due to grave, genuine circumstances, please reach out to the Prof, and we will consider your case. While academic commitment is appreciated, please put your and your family's health first :)

For BASH, since it is autograded, please make sure that you test your code by running it from within your Docker environment. If you use Windows, there might be an issue with the newline character, so you could use an editor like vim on the Docker command line. *In any case, please check on Docker before submitting!*

For some scripts, if you use Windows, you might see an error like "\r: Command not found" when you try to run stuff on Docker. This is because Windows and Unix encode newlines in text files differently: it's "\r\n" for Windows, and "\n" for Unix. Simply

running `dos2unix <filename>` should fix the issue, and we will be mindful of this while grading. You can `apt-get install dos2unix`

Dear Rushabh,

Times New Roman is way classier than Arial.

Sincerely,
Mihir