Lab Assignment 3

Duration: 3 weeks

1. Create (somewhere on your computer) a new empty folder named Inf43Hw2. In that folder, create a plain text file with your name and student ID # on a single line. Don't use MS Word for this, use a text editor, for example Notepad or TextEdit (you may need to choose Plain Text under Format or Preferences). Save the file as file1.txt.
2. In your Git shell, navigate to Inf43Hw2. Use the **cd** command to change your current folder/directory. Note that the Windows Git Bash shell follows Unix/Linux shell conventions so if you're on Windows, you still need to use Linux-style paths with forward slashes (e.g., c:\my_folder\my_subfolder would be /c/my_folder/my_subfolder). Linux commands like ls, pwd, and grep should all work in the Git Bash shell.
3. Note: At this point you may find that Git wants you to tell it your name and email address. You can do this with two commands like these:
    - **git config --global user.name "nidhi"**
    - **git config --global user.email "nidhi.2592@gmail.com"**
    - To look at all your configuration information: **git config --global -1**
4. Create a local Git repo by running the command **git init**.
5. Run **git status**. Note that file1.txt is listed as untracked. We want Git to track it, so run **git add file1.txt**. When you "add" a file you are telling Git to keep track of it. "add" also tells Git to *stage* the file, which means put it in the stage of being ready to be committed.
6. Run **git status** again. Note that file1.txt is now listed as a file to be committed (i.e., it's staged).
7. Let's commit file1.txt to our repository. Run **git commit -m "Committing a new file with my name"**. When you "commit,", you in effect copy all staged files to the repository. The "-m" is a flag (that's what the hyphen indicates) which tells Git that the following string is a message to record with the commit.
8. Run **git log**. This will display the history of changes made to the repository. The one and only entry will be for the commit of file1.txt you just did.
9. Edit file1.txt and change the spelling of your name to something incorrect. Save file1.txt with the error. (This small error stands in for a long complex series of edits that you want to undo.)
10. Run **git reset --hard**. "reset --hard" removes all uncommitted changes, so all files in the repo will return to their contents as of the latest commit. There are many ways to undo changes in git, and "reset --hard" is generally considered to be dangerous. Look at file1.txt and observe the effect of reset --hard.
11. Edit file1.txt to *remove* your student ID# and *include* the name of your major, and save the file.
12. Commit with **git commit -m "Now has my major"**. This doesn't work. Git tells you there are "changes not staged for commit".
13. Try again with **git commit -a -m "Now has my major"**. The power of the "-a" flag is that it tells git to automatically stage all tracked, modified files before the commit.
14. You can also explicitly stage a file. Add the name of your favorite restaurant and favorite movie to to file1.txt, save it, and run **git stage file1.txt**. Now run **git commit -m "Added favorite restaurant"** to commit. "git stage" is really just another name for "git add".

15. You set the commit message to "Added favorite restaurant", but the file also includes your favorite movie, so maybe we should have included that in our commit message. Amend your commit message with **git commit --amend -m "Added favorite restaurant and movie"**.
16. Run **git log** to make sure you have successfully changed history.
17. You removed your student ID# a few steps back. Let that edit stand in for deleting, a few months ago, a block of code that you now want to examine. git will help you go back in time. Note that each commit has long, seemingly random, string of hexadecimal digits associated with it. This is called a "hash" and is a unique identifier for the commit. Find the hash associated with the "Committing a new file with my name" commit. Run **git checkout xxxx**, replacing **xxxx** with the first four digits from that hash (thankfully typing in the entire hash is not required). You will see a frightening message about a detached HEAD.
18. git can keep track of separate, parallel, streams of edits to a project. Each stream of edits is called a branch, and a branch can have a name. For instance, multiple programmers who are working on and commiting changes to the same file will probably establish different branches. HEAD is git-ese for the current (not necessarily the last) commit in the current branch. Since we've gone back in time and are potentially (but haven't yet) starting a new branch, HEAD is "detached" (from any established, named branch). Ouch!
19. Take a look at file1.txt and note the later-deleted Student ID#. Now to return to the present: **git checkout master**. "master" is the name of the default branch created when the repository was made. Look at file1.txt again. Run **git log** again and you'll see it has the same three commits.
20. Create a new text file called file2.txt that contains your expected graduation year and first job title on a single line.
21. Stage file2.txt, and then commit it with a useful message.
22. Run **git log**. Notice that you see log entries for both commits that you've performed.
23. Run **git log file2.txt**. Notice that you only see the log entry involving file2.txt.
24. Modify file1.txt to have the name of your favorite color on a new line.
25. Delete file2.txt.
26. Run **git status**. Note that file2.txt is listed as deleted. Also note that the status information helpfully says "git add/rm ..." to update what will be committed.
27. You use the **git add** command to stage a new or modified file. However, to stage the deletion of a file, you need to use the **git rm** command. So run the commands **git add file1.txt** and **git rm file2.txt** to set the stage.
28. Commit the changes with the commit message "Deleting file2.txt".
29. Run the command **git log -p -3**. The -p flag will show you the diffs for each change. The -3 will limit what's displayed to the last 3 log entries. Take a few minutes to look carefully at the output log and see if you can figure out how to interpret it.
    - Note: A command such as **git log** sends text output to the shell using a bash command named "less" to display one windowfull of output at a time ("less" is named after a similar, earlier utility named "more", in a classic example of hilarious techie humor). At the **:** prompt, you can press **h** for help, **q** to exit, **Enter** to advance one line, or **Space** to advance one screenfull.
30. Now you decide you actually wanted to keep file2.txt, but you deleted it! Fortunately, you had added it to git, so you can still get it back. There are several ways to do this. The simplest is probably to use the command **git checkout HEAD~1 file2.txt**. What does this do? HEAD represents the most recent commit or snapshot. ~1 tells Git to go back one version from the most recent snapshot (i.e., HEAD). In this older snapshot, file2.txt still existed, and checkout

tells Git to retrieve it. If you now look in your folder, you'll see file2.txt. And is file1.txt changed?

31. Run **git add file2.txt** to stage file2.
32. Run **git status**. Note that file2.txt is staged. Commit it with the commit message "Re-adding file2.txt".
33. Run the command **git log -p** to see all of the log entries.
34. Now run the command **git log -p > git_log_partB.txt**. (The > is a shell command that redirects the output of the program on >'s left to the file named on >'s right.)
35. Open git_log_partB.txt. It should like just like the output you saw for step 33. If you're on Windows and viewing it in Notepad, the spacing will probably look wrong, so try opening it in a different text editor (like Wordpad or Notepad++).
36. Add all files to GIT Hub, by using GIT bash. Practice fetch, merge, push and pull in your created repository.