# CS 242: Lab 4 Problems: (Git)

Note: The lines starting with a ‘$’ symbol represents a bash command line.

1. ‘Git’ is a Version Control System (VCS) or Revision Control System; a VCS is a software that maintains a ‘change history’ of your files. For a particular file, the change history keeps track of all the changes made on the file and for each of those changes:
   1. who made the change,
   2. what changes are made, and
   3. when the change was made.
2. Git is currently the most widely used VCS. Before Git, came other VCSs. Let us learn about some of them before we dive deep into the world of Git.
   1. SCCS (Source Code Control System) is the first VCS. It was developed by Marc Rochkind at Bell Labs, USA, in 1972. The motivation was to facilitate UNIX programming on IBM mainframe computers.
   2. CVS (Concurrent Versions System): In 1986, came CVS. It overcame one of the limitations of SCCS. The limitation is that, in order to edit a file, SCCS requires you to get a lock on it. Once you got the lock, only you can edit the file. Once the editing is done, you have to unlock the file so that other can edit it further. CVS resolves it this way: when you want to edit a file, you get a copy of that file (not the original file). Then you can edit that copy and merge your changes back to the original file. Therefore, if anyone else wants to edit the file while you are editing it, then he/she gets another copy, edits it and merges it back. If your edits and the other developer’s edits conflict, then you can resolve the conflicts by going through them manually.
   3. Git: In 2005, enters Git. Guess who developed it? None other than Linus Torvalds, who created Linux. Linus addressed a limitation that both of SCCS and VCS have. The limitation is that they were ‘centralised’; meaning that the original files and change history are always kept in a central location, such as a server. Therefore, if anything happens to that server, then all original files and change history would be lost forever. To overcome this problem, Linus made Git ‘distributed’: the original files and change history are mirrored in everyone’s computer. (By the way, there is another distributed VCS named ‘Mercurial’. It is also very popularly used. Google up ‘Mercurial’ in your spare time.)
3. Now, let us dive deep into the world of Git. For starters, install it in your computer using the following commands:  
   $ sudo apt-get update  
   $ sudo apt-get install git
4. Once Git is installed, then let us start by creating a new project directory. A project directory is nothing but a directory inside which you keep all the files and sub-directories related to that project.  
     
   ## Create a project directory   
   $ mkdir Test-proj

## Enter the directory  
$ cd Test-proj

1. Then convert the project directory into a Git repository (in short, ‘repo’):  
   $ git init  
   ## The aforementioned command creates a sub-directory named ‘.git’ under your project directory. Inside the ‘.git’ sub-directory, git stores the change history.
2. Now, create a file for this project and a line to it:  
   $ echo ‘This is the first line.’ > prog.c
3. ‘Stage’ the newly created file.  
     
   ## Stage ‘prog.c’ i.e. add it to Git’s index. Thus, Git takes a ‘snapshot’ of the changes made on this file.  
   $ git add prog.c  
   ## Once the changes of a file is snapshotted, the file is said to be ‘in staging area’ or the file is ‘staged’ to be committed (but not committed yet).  
   ## Instead of staging one file at a time, you may also use ‘$ git add . ’ which will stage all files inside the current directory.  
     
   ## Check Git status  
   $ git status  
   ## This command returns the following output:

|  |
| --- |
| On branch master  Initial commit  Changes to be committed:  (use "git rm --cached <file>..." to unstage)  new file: prog.c |

Now, let us try to understand the output.

* 1. At the top, it says, ‘On branch master’; a ‘branch’ in Git represents a single line of development. By default, Git creates a branch named ‘master’. However, you can create as many branches as you need. Usually, a professional developer uses three branches: master, bug-fix and dev.
     1. The developer uses the master branch to ‘release’ the files he/she wants to share with the users.
     2. The bug-fix branch is used to fix the bugs reported by the users.
     3. The dev branch is used for further development, like - adding new features requested by the users.

Please note that, apart from ‘master’, the other branch names are user-defined. Names like ‘bug-fix’ and ‘dev’ are simple conventions followed by a large number of developers. You may create branches with the names of your choosing. At the same time, you may create as many branches as you need. Thinking how to create branches -- we will learn that very soon. For the time being, let us go back to the ‘$ git status’ output.

* 1. Next, it says that the ‘Changes to be committed:’ are ‘new file: prog.c’, which you have just created. It also suggest that if you wish to unstage ‘prog.c’, you may do so using the following command:  
     $ git rm --cached prog.c

1. Do you notice that all Git commands follow the syntax of ‘$ git <command>’. If you wish to read the manual of a particular command, use ‘$ man git <command>’. For example,  
   $ man git add  
   ## The aforementioned command opens the manual page of ‘git add’  
     
   If you are extremely cautious and want to know what a particular command will do to your repo, without actually running the command, use a ‘dry run’ of the command:   
   ‘$ git <command> <options or flags> --dry-run’.  
   For example, try a dry run of unstaging prog.c:  
   $ git rm --cached prog.c --dry-run  
   ## Due to the '--dry-run' option, no changes are done. But what will be done during an actual run is printed in the text format.
2. ‘Commit’ the changes. In other words, move the snapshotted changes from the staging area to the ‘commit history’ of the repo.  
   $ git commit -m “your message that briefly describes why you made the change”  
   For example,  
   $ git commit -m “Added the first line”  
   ## The double quotes are mandatory around the message. You may also use single quotes instead of the double ones.  
   Note: While running git commit, Git might ask you for your email and name. Please fill them in.   
   $git config --global user.email "[you@i](mailto:you@example.com)itg.ernet.in"  
   $ git config --global user.name "Your Name"  
     
   Once you successfully run the git commit command, the output is:

|  |
| --- |
| [master (root-commit) ab025ed] Added the first line.  1 file changed, 1 insertion(+)  create mode 100644 prog.c |

Here, the string ‘ab025ed’ signifies the ‘commit state’ aka ‘commit hash’.  
 Your computer may produce a different commit hash.   
 But for a particular repo, each commit will have a unique hash.  
 Please note that the shown commit hash is only the first few characters of the full commit hash. You can see the full commit hash by issuing the command:  
 $ git rev-parse HEAD  
 Here, ‘HEAD’ is a pointer aka ‘ref’ that points to the current state you are working in.  
 The shown commit hash is basically a short version of the full commit hash.  
 Therefore, you can print the short commit hash by:  
 $ git rev-parse --short HEAD  
 The short commit hash is the one that is shown when you make a commit.

## Re-check the status  
$ git status  
## The output is:

|  |
| --- |
| On branch master nothing to commit, working directory clean |

1. Now, committing should be sufficient if you wish to have your repo on a single device, such as your lab PC. However, if you are working from multiple devices, or working with multiple team members, then you need to mirror your ‘local repo’ (local to a particular device) in a centralised location. Here, comes the utility of host websites, like - GitHub. To mirror your local repo in GitHub, please follow along:
   1. First, create a free user account in GitHub: <https://github.com/> .
   2. Then create a remote repo following the instructions at: <https://help.github.com/en/articles/create-a-repo>
      1. Use the same name for your remote repo as you local repo i.e. ‘Test-proj’. The name matching is not mandatory. But, most developers follow this convention for easier memorability.
      2. Un-select ‘Initialize this repository with a README’ checkbox, if it is selected by default. Because you do not want to directly create files inside the remote repo. You only want to create files in your local repo and then mirror them in the remote repo.
   3. Once your remote repo is ready, you will be redirected to a ‘Quick setup --’ page. Please go to section titled ‘…or push an existing repository from the command line’. It tells you how to mirror aka ‘push’ your local repo to this remote repo. The first command is:  
      $ git remote add origin <https://github.com/><your git username>/Test-proj.git  
      This command tells your remote repo about the location of your remote repo. The remote repo is called the ‘origin’ since all your devices and your team members will sync their local repo with the remote repo.  
        
      Once the origin is added, you simply push the committed changes to the origin.  
      $ git push -u origin <branch-name>  
      Since, you have made the changes in your ‘master’ branch, the command will look like:  
      $ git push -u origin master  
      This command will ask for your GitHub credentials. Please fill them in.
2. Now, take a breather. We just learned the commands that we will use most often with Git.   
   Now, let us explore some of the fancy commands that we want to use less frequently. But, there will be circumstances when we will need them.
3. Un-do the changes you have made to an un-staged (but ‘tracked’) file:  
   $ git checkout -- <filename>  
   Usually, a file is ‘tracked’ by Git if it is: (a) staged at least once, and (b) not included in ‘.gitignore’. ‘.gitignore’ is a file where you enter the names of the files you do not want Git to track. We will learn how to create a ‘.gitignore’ file very soon.  
   For now, let us see how ‘git checkout’ works. Suppose, we create a new file inside a Git repo.  
   $ echo "This is a new file." >> newfile.txt  
   Then we try to checkout this file.  
   $ git checkout newfile.txt  
   And the output is:

|  |
| --- |
| error: pathspec 'newfile.txt' did not match any file(s) known to git. |

The error is thrown because the file is unknown to Git. Since, the file is new and never been staged, it is un-tracked by Git. So, let us first stage it.  
$ git add newfile.txt  
Now, the file is tracked by Git and its contents are staged. Therefore, let us make some changes to its contents.  
$ echo "This is another line in the new file." >> newfile.txt  
Let us check whether the change is reflected inside the file.  
$ cat newfile.txt

|  |
| --- |
| This is a new file. This is another line in the new file. |

This change is not yet staged. But the file is tracked. Therefore, ‘git checkout’ should be able to discard the un-staged change i.e. remove the second line of the file.  
$ git checkout newfile.txt  
$ cat newfile.txt

|  |
| --- |
| This is a new file. |

Hurray! Thus, ‘Git checkout’ discards un-staged changes in a tracked file.

1. Un-stage a staged file.  
   $ git reset HEAD -- <filename>
2. To see a log of all changes in your local repo:   
   $ git log  
     
   To see a shorter version of the log, use:   
   $ git log --oneline
3. A situation may arise when there are changes in the origin that are not in your local repo, but all changes in your local repo are in origin. This situation is called ‘the origin is ahead of your local repo’. Then you need to download or ‘pull’ the newer changes into your local repo. Such a situation happens when your team members have made some updates in the origin. In that case, use:  
   $ git pull origin <branch-name>
4. Remember, we talked about creating branches other than the ‘master’. You can create a new local branch in your local repo using:  
   $ git <branch-name>

List all local branches:  
$ git branch  
  
Switch to a particular branch:  
$ git checkout <branch-name>  
  
Merge ‘branch-A’ to ‘branch-B’ (not ‘branch-B’ to ‘branch-A’):  
$ git checkout branch-B  
$ git merge branch-A  
If there are merge conflicts, you have to manually resolve it. You may use software like GitKraken to make merging easy; please check out Manan Gupta’s lecture slides for that.

1. You can ‘clone’ someone else’s repo into your local device:  
   $ git clone <repo-url>  
   For example, try:  
   $ git clone <https://github.com/sap01/Test-proj.git>
2. You can tell Git never to track a specific file, say – prog.o, or a particular types of files, like - \*.o :  
   $ touch .gitignore  
   $ echo ‘\*.o’ >> .gitignore  
     
   ‘.gitignore’ is a special file which tells Git which files to ignore. There are some useful .gitignore templates specific to programming languages at: <https://github.com/github/gitignore> . You can download and use them directly or modify them and then use the personalised version. For example, the C language project template is at: <https://github.com/github/gitignore/blob/master/C.gitignore>
3. What if you are working on something and made some changes after the last commit; then suddenly your priorities changed – maybe your team member asked for some other changes; so, now you have to work on something else; but, you can not let go the changes you have so painstakingly made. In that case, you ‘stash’ your changes temporarily:  
   $ git stash  
     
   After stashing, you are back to the last committed state. Now, you're free to make changes, create new commits, switch branches, and perform any other Git operations; then come back and re-apply aka ‘pop’ your stash when you're ready:  
   $ git stash pop  
     
   Warning: Git stash does not stash ignored and un-tracked files. There are ways to do that, but that is out-of-scope for this problem-set.
4. Finally, there are so many commands to remember! So, create your own cheat-sheet and paste it near you.   
   And ‘Git things done!’