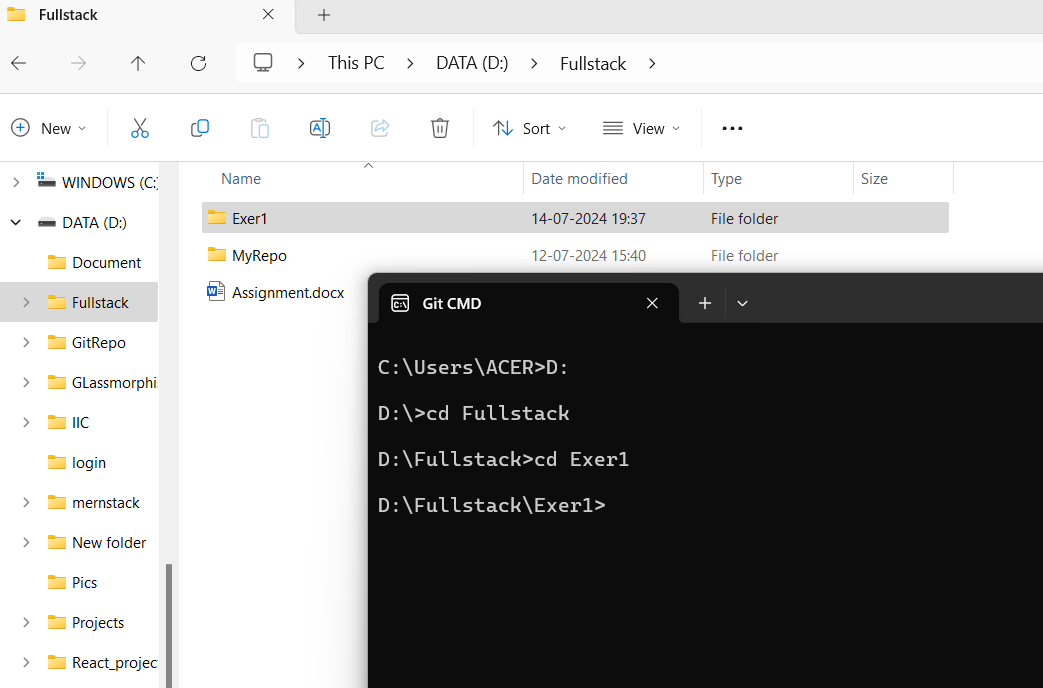
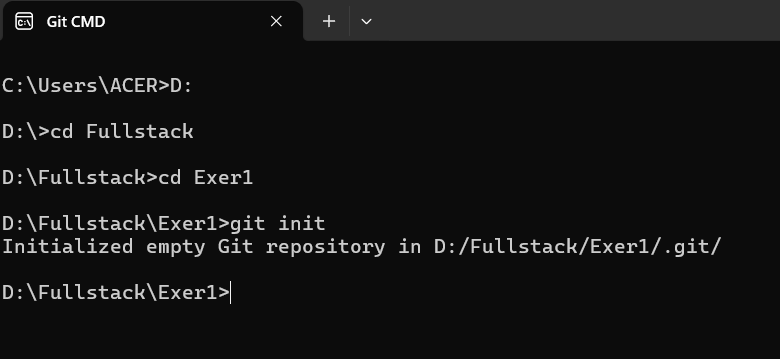
ASSINGMENT-2



**Exercise 1**

1. Create a new directory and change into it.

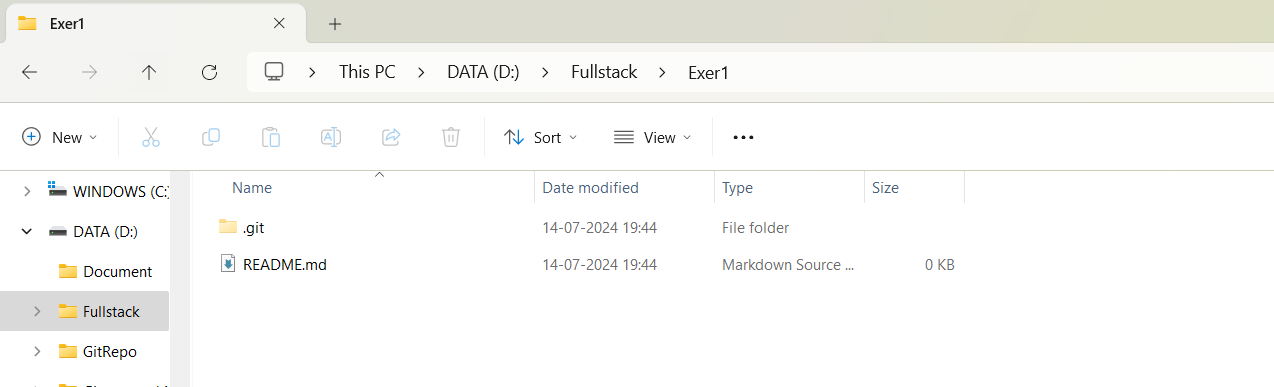


1. Use the init command to create a Git repository in that directory.
2. Observe that there is now a .git directory.

A screenshot of a computer

Description automatically generated

1. Create a README file

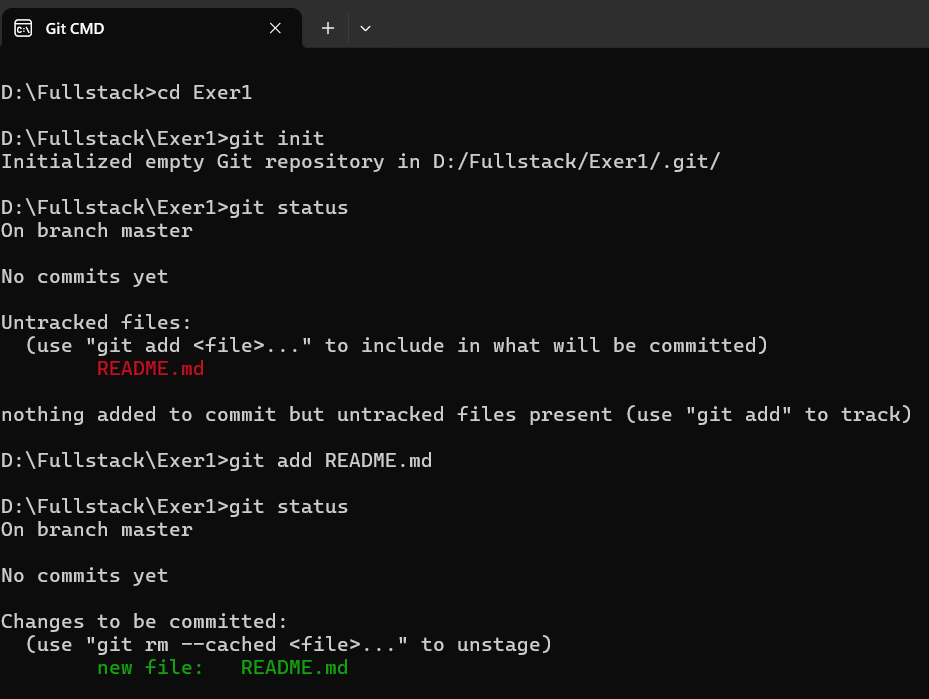


1. Look at the output of the status command; the README you created should appear as an untracked file.

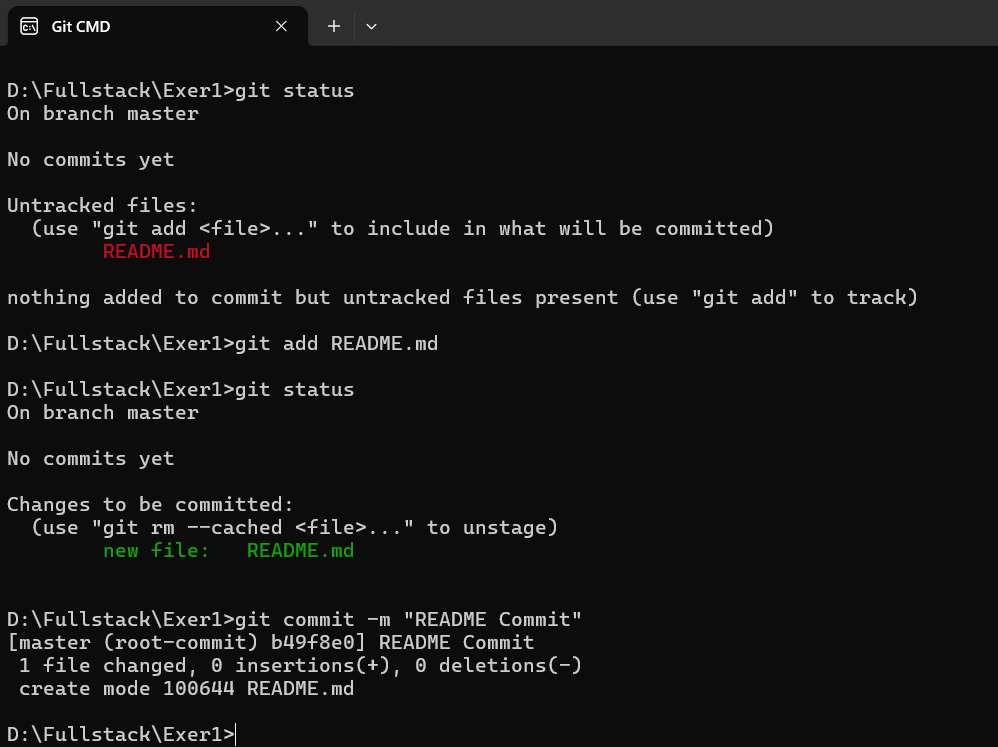
A screenshot of a computer

Description automatically generated

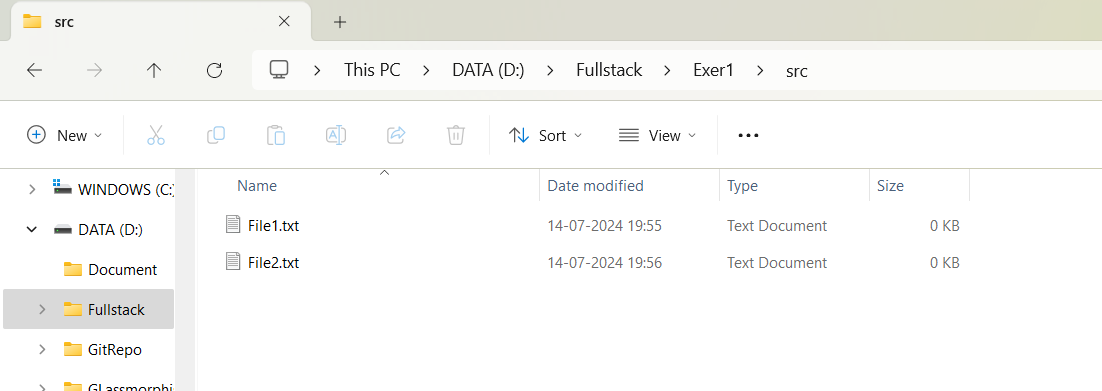
1. Use the add command to add the new file to the staging area. Again, look at the output of the status command.



1. Now use the commit command to commit the contents of the staging area.

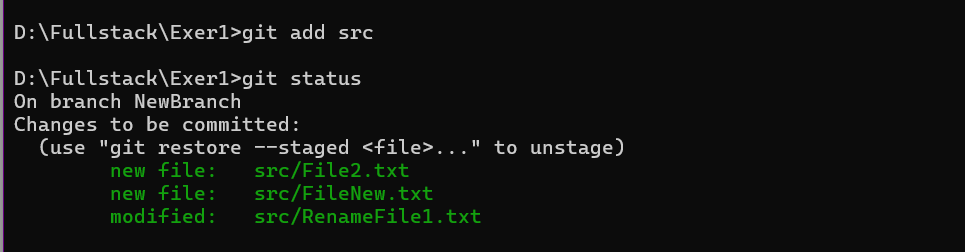


1. Create a src directory and add a couple of files to it.

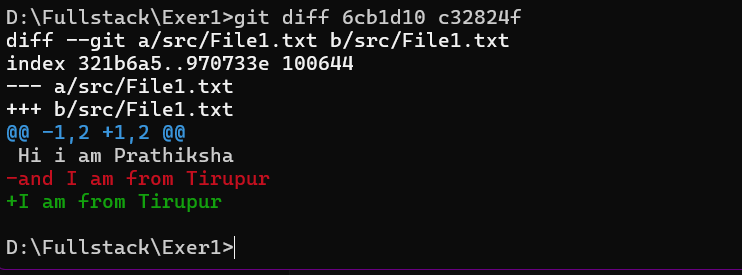


1. Use the add command, but name the directory, not the individual files. Use the status command. See how both files have been staged. Commit them. (Git add src)

A screenshot of a computer

Description automatically generated  
  


1. Make a change to one of the files. Use the diff command to view the details of the change



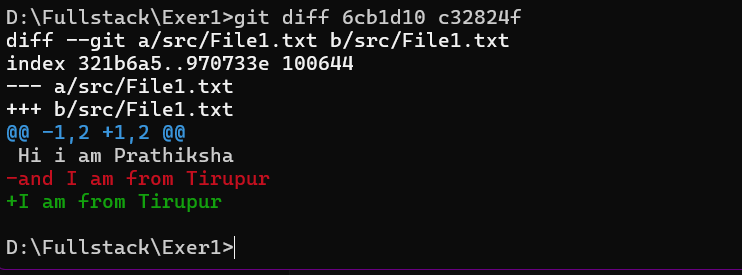
1. Next, add the changed file, and notice how it moves to the staging area in the status output. Also observe that the diff command you did before using add now gives no output. Why not? What do you have to do to see a diff of the things in the staging area?

(Git diff –staged)

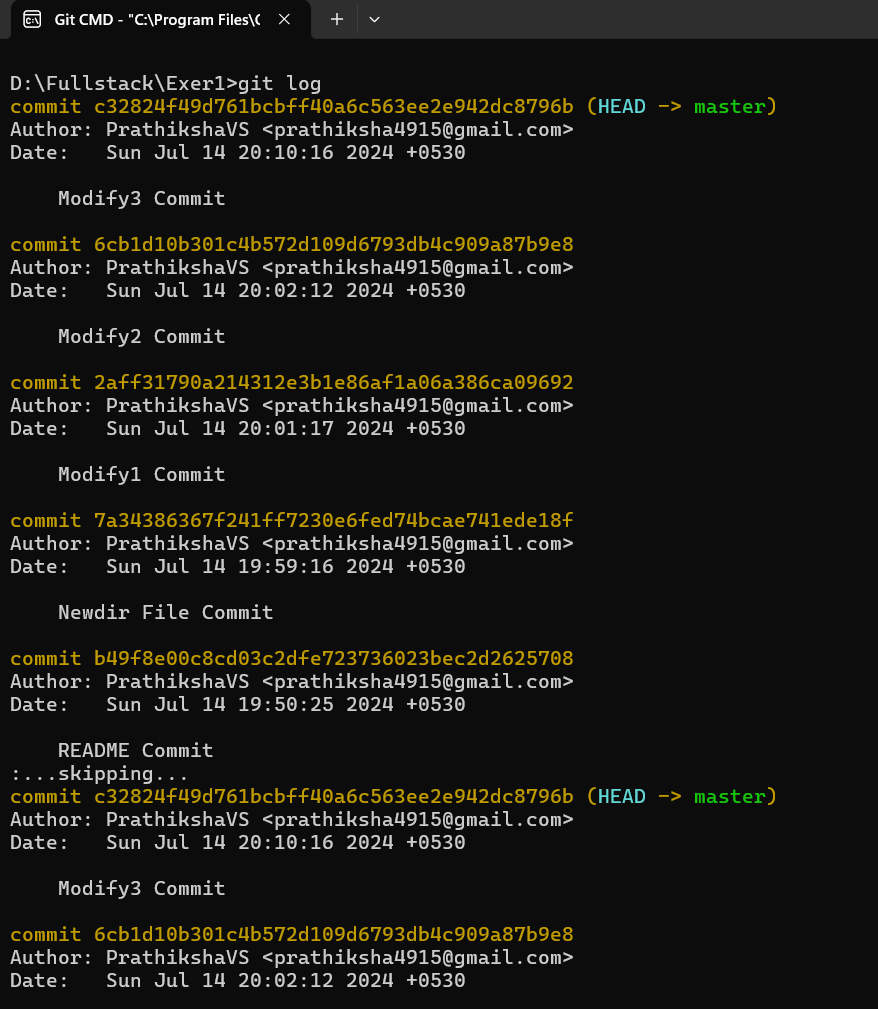
A computer screen with white text

Description automatically generated  
 We have to commit and using the commit id or two files after the completion of commit to see the changes in staging area.

1. Now – without committing – make another change to the same file you changed in step 10. Look at the status output, and the diff output. Notice how you can have both staged and unstaged changes, even when you’re talking about a single file. Observe the difference when you use the add command to stage the latest round of changes. Finally, commit them. You should now have started to get a feel for the staging area.



1. Use the log command in order to see all of the commits you made so far.



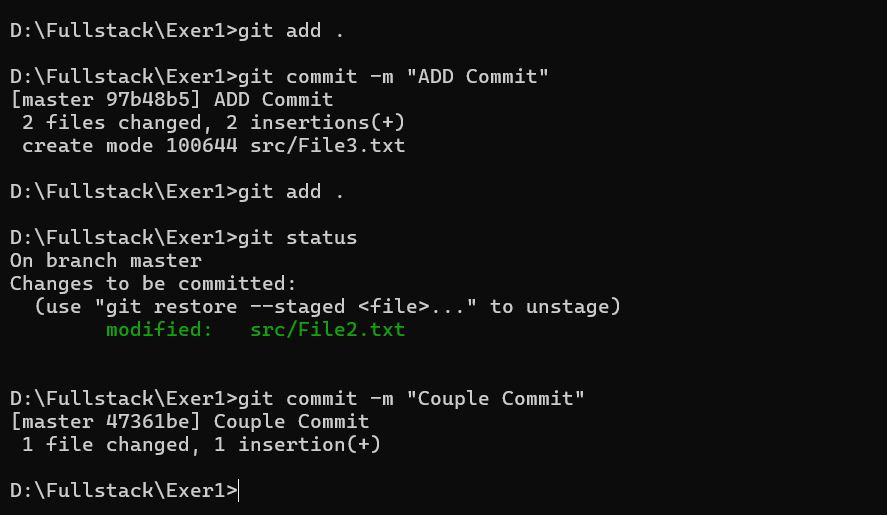
1. Use the show command to look at an individual commit. How many characters of the commit identifier can you get away with typing at a minimum?

A computer screen shot of a program

Description automatically generated

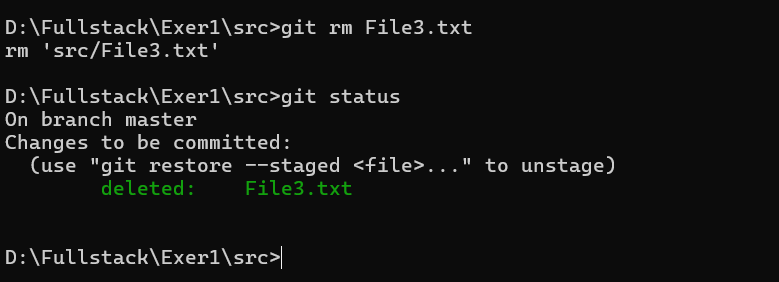
The minimum characters are four but it has some risk so we have to give Atleast minimum7-8.

1. Make a couple more commits, at least one of which should add an extra file.

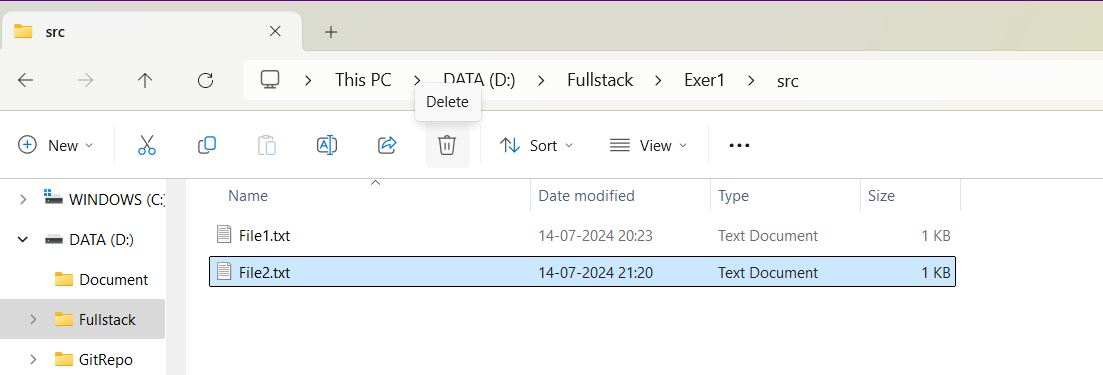


**Stretch Task**

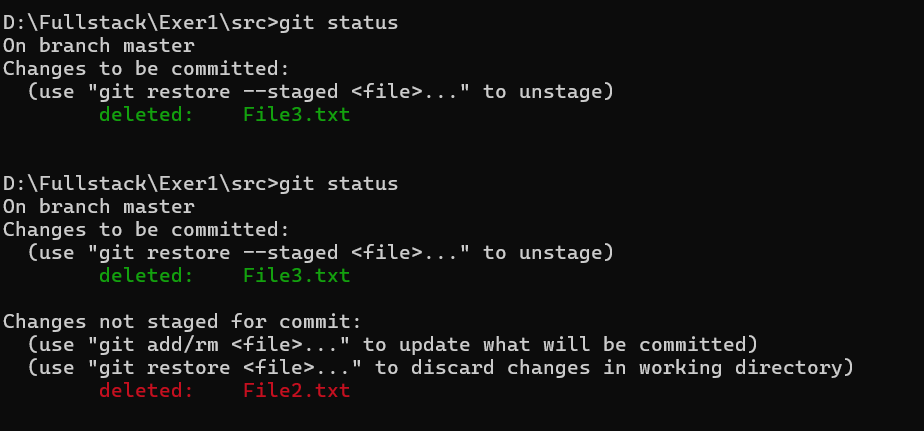
1. Use the Git rm command to remove a file. Look at the status afterwards. Now commit the deletion.



1. Delete another file, but this time do not use Git to do it; e.g. if you are on Linux, just use the normal (non-Git) rm command; on Windows use del.

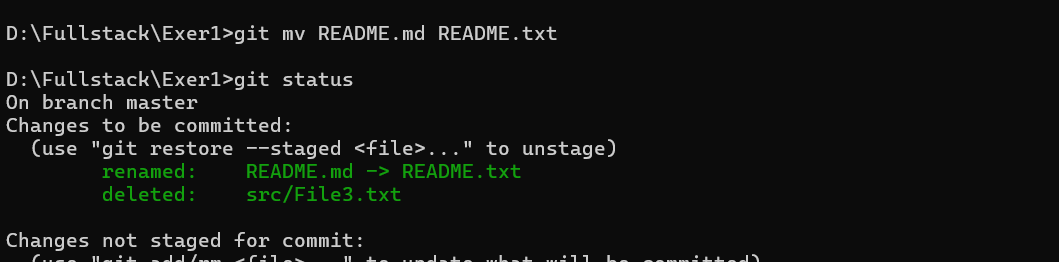


1. Look at the status. Compare it to the status output you had after using the Git built-in rm command. Is anything different? After this, commit the deletion.

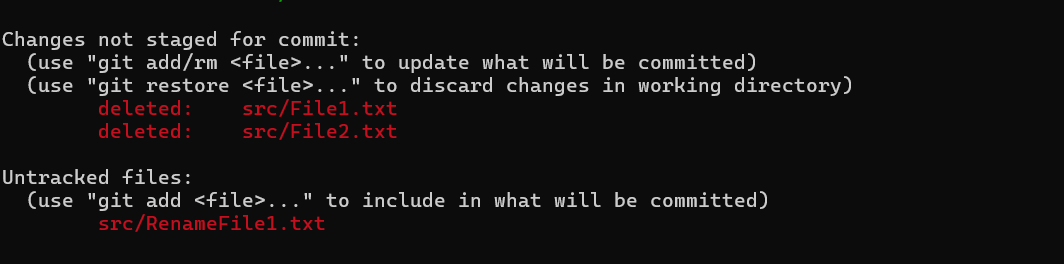


The changes were done in working area so the changes was untracked now before it was tracked

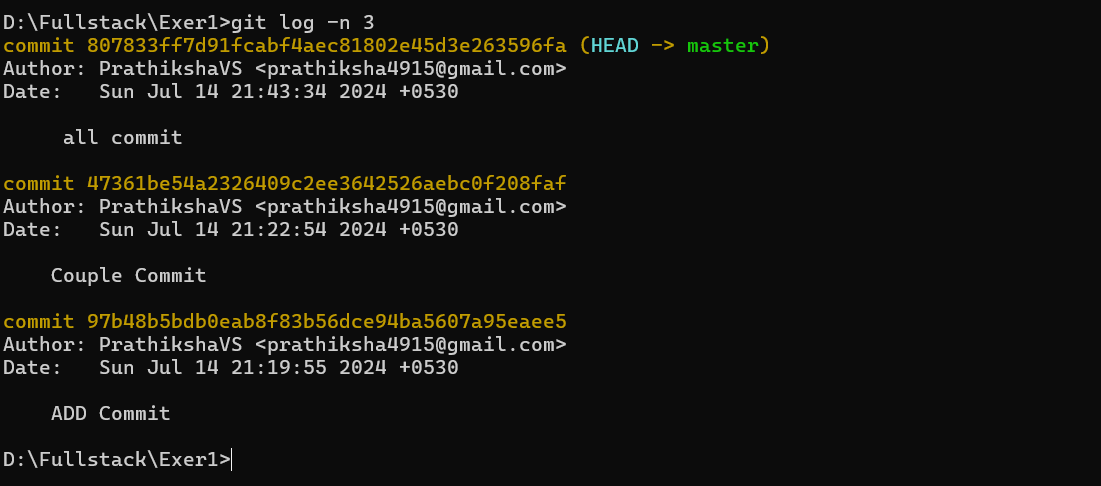
1. Use the Git mv command to move or rename a file; for example, rename README to README.txt. Look at the status. Commit the change.



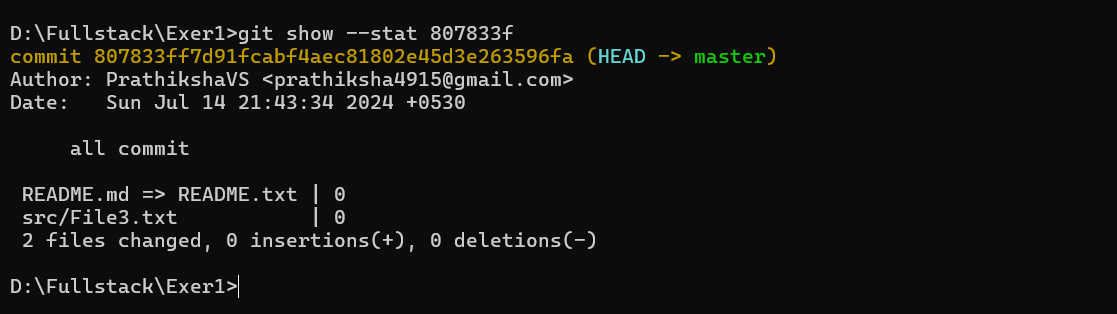
1. Now do another rename, but this time using the operating system’s command to do so. How does the status look? Will you get the right outcome if you were to commit at this point? (Answer: almost certainly not, so don’t. ) Work out how to get the status to show that it will not lose the file, and then commit. Did Git at any point work out that you had done a rename?



1. Use git help log to find out how to get Git to display just the most recent 3 commits. Try it.



1. If you don’t remember, look back in the slides to see what the --stat option did on the diff command. Find out if this also works with the show command. How about the log command?

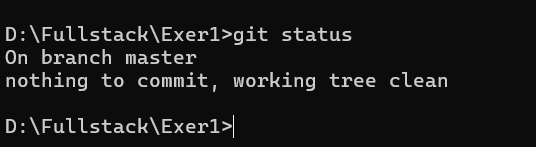


1. Imagine you want to see a diff that summarizes all that happened between two commit identifiers. Use the diff command, specifying two commit identifiers joined by two dots (that is, something like abc123..def456). Check the output is what you expect.

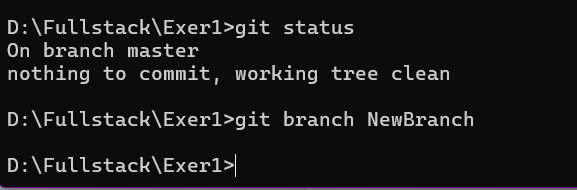


**Exercise 2**

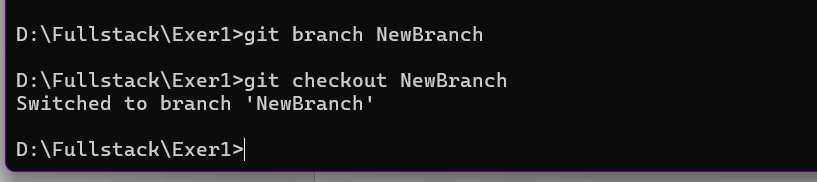
1. Run the status command. Notice how it tells you what branch you are in.



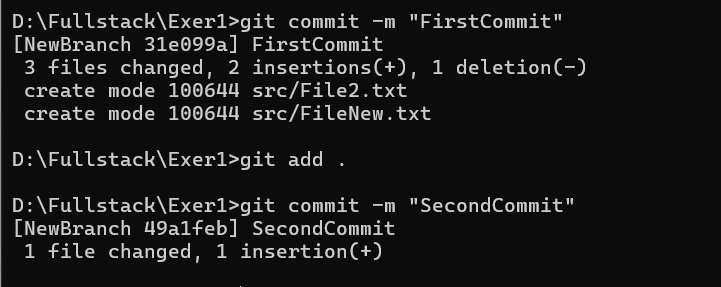
1. Use the branch command to create a new branch.



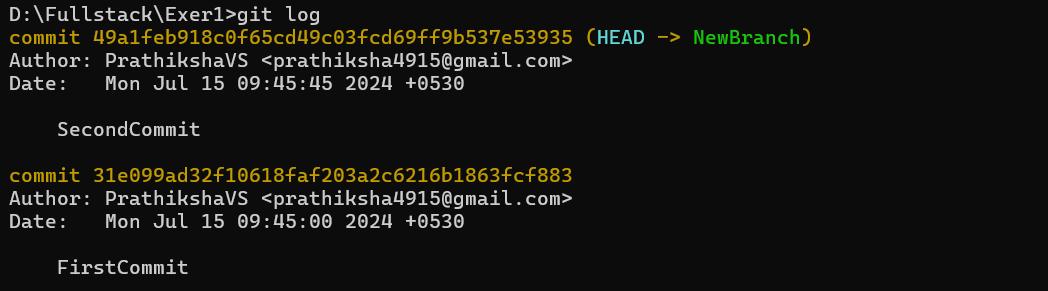
1. Use the checkout command to switch to it.



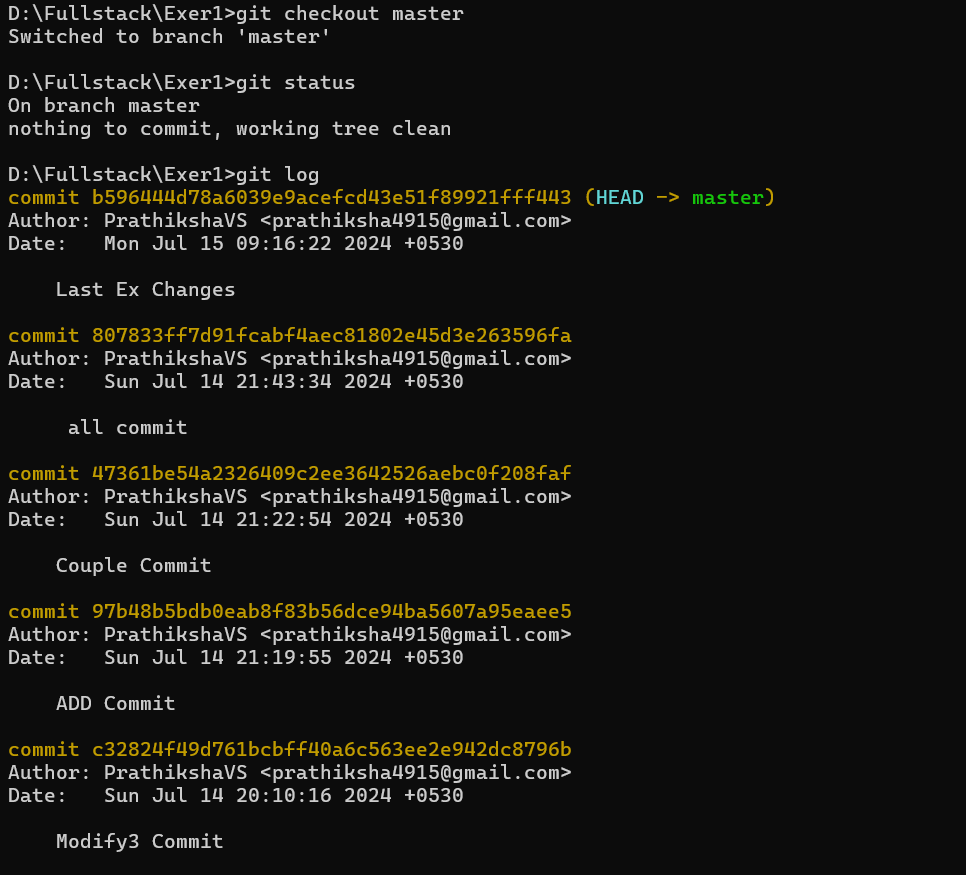
1. Make a couple of commits in the branch – perhaps adding a new file and/or editing existing ones.



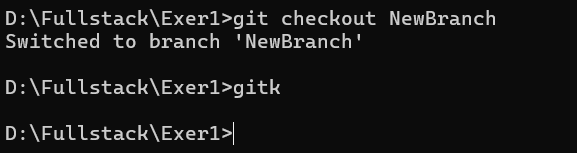
1. Use the log command to see the latest commits. The two you just made should be at the top of the list.

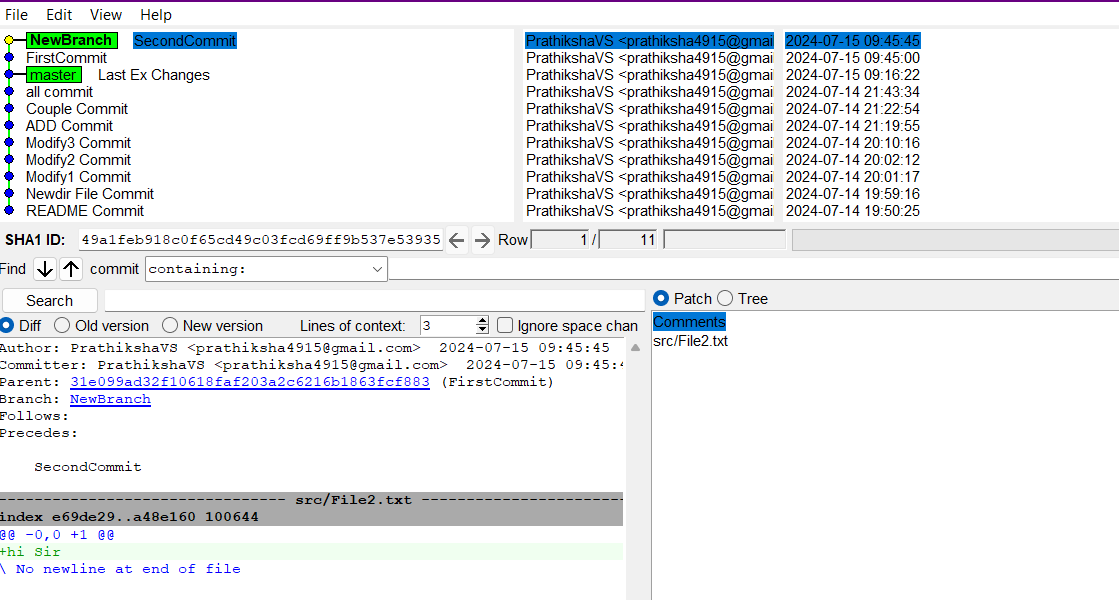


1. Use the checkout command to switch back to the master branch. Run log again. Notice your commits don’t show up now. Check the files also – they should have their original contents.

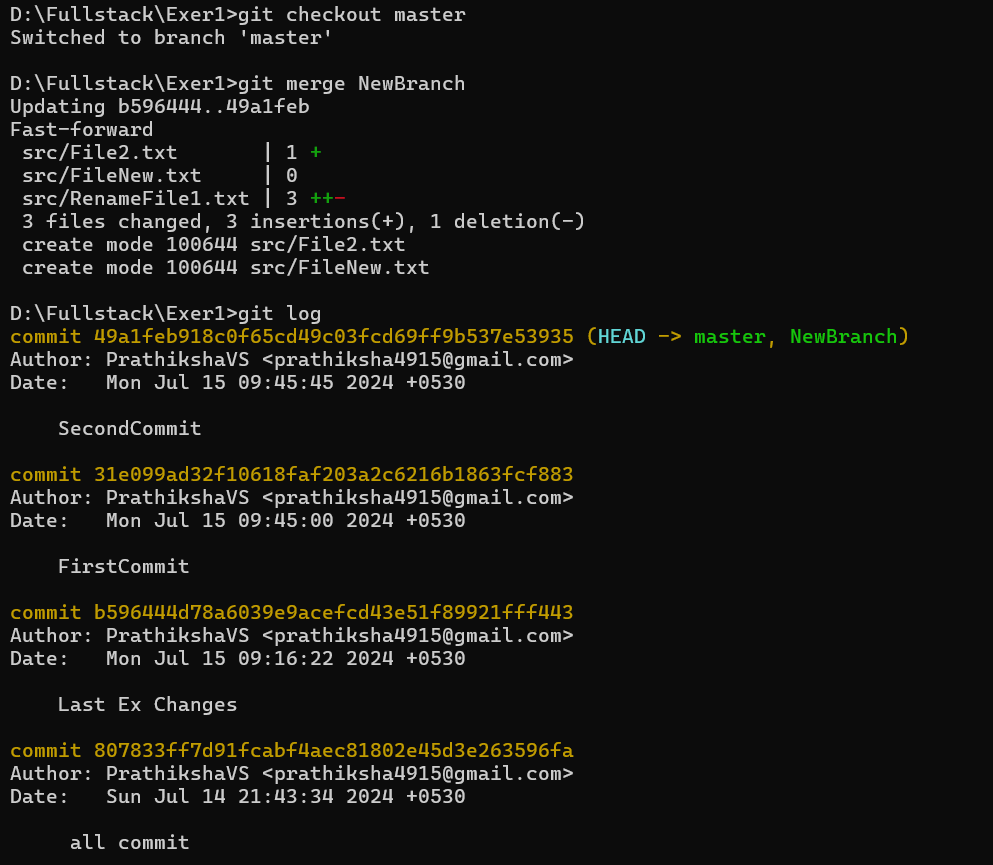


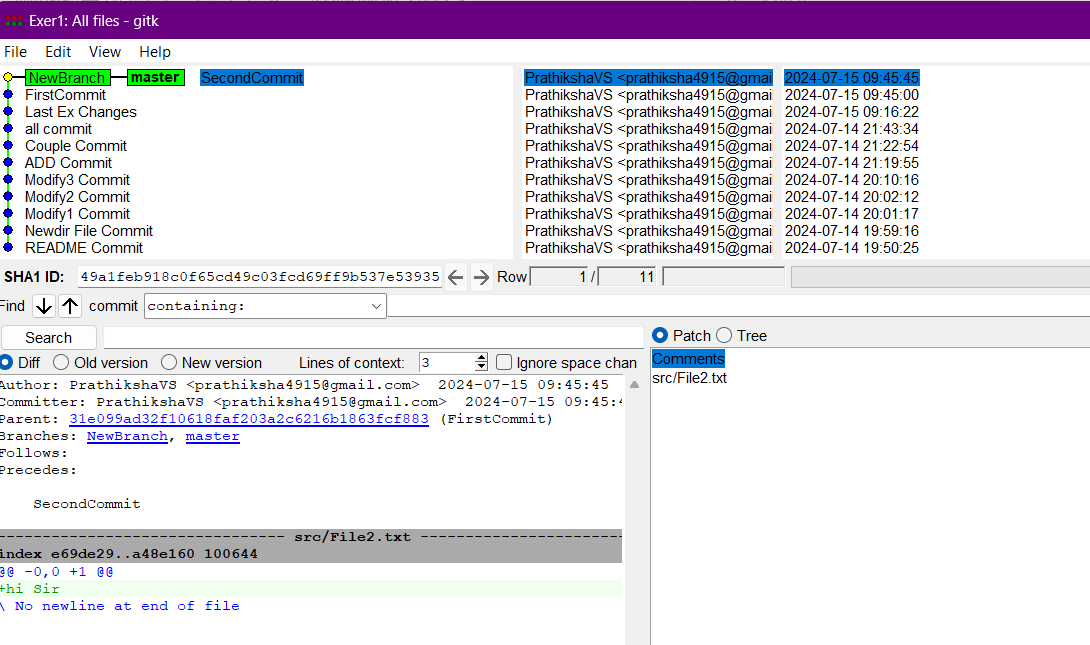
1. Use the checkout command to switch back to your branch. Use gitk to take a look at the commit graph; notice it’s linear.



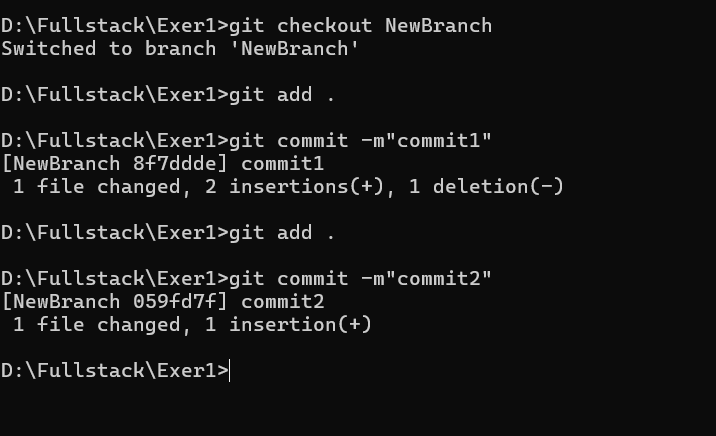


1. Now **checkout** the master branch again. Use the **merge** command to merge your branch in to it. Look for information about it having been a fast-forward merge. Look at **git log**, and see that there is no merge commit. Take a look in **gitk** and see how the DAG is linear.

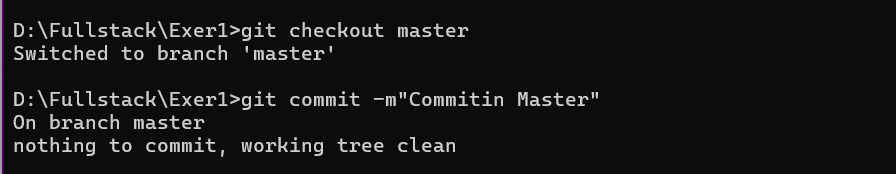




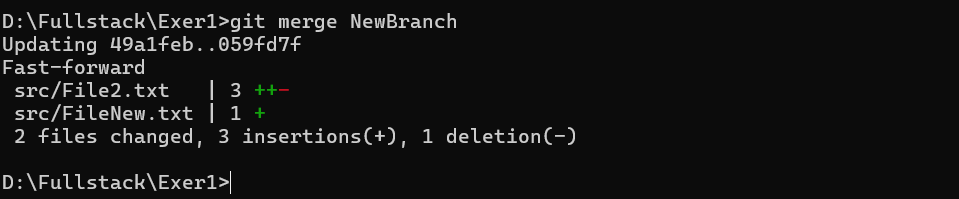
1. Switch back to your branch. Make a couple more commits



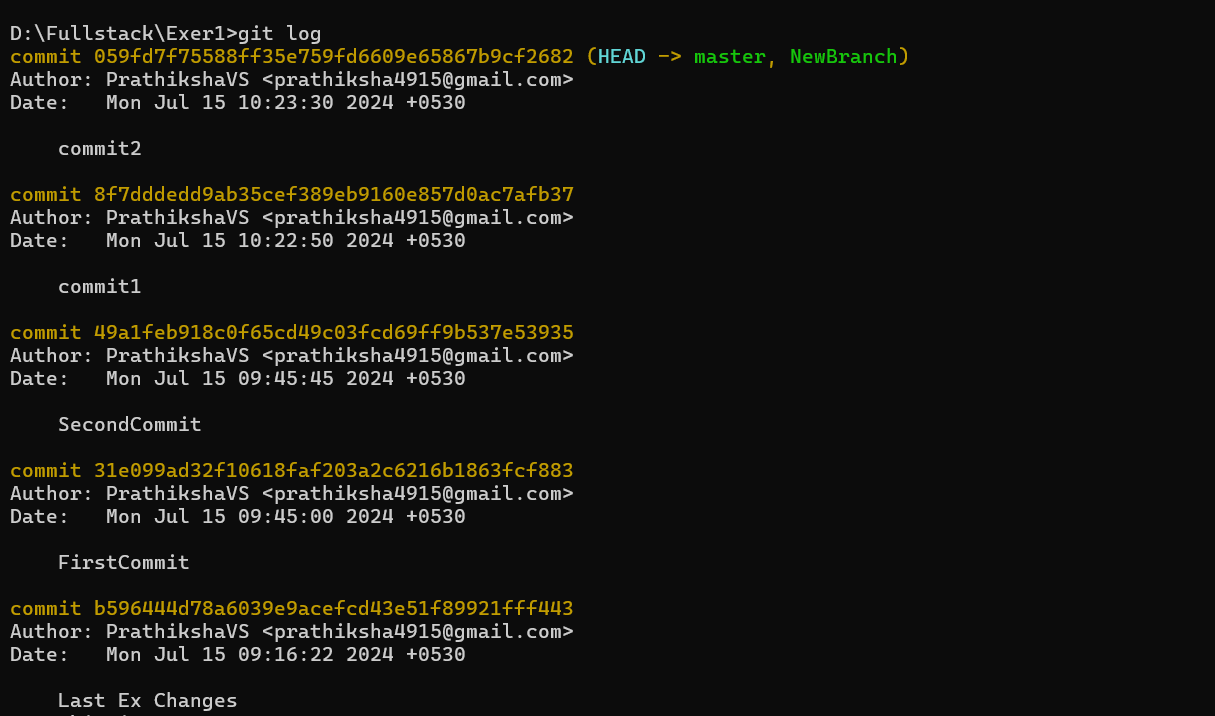
1. Switch back to master. Make a commit there, which should edit a different file from the ones you touched in your branch – to be sure there is no conflict.

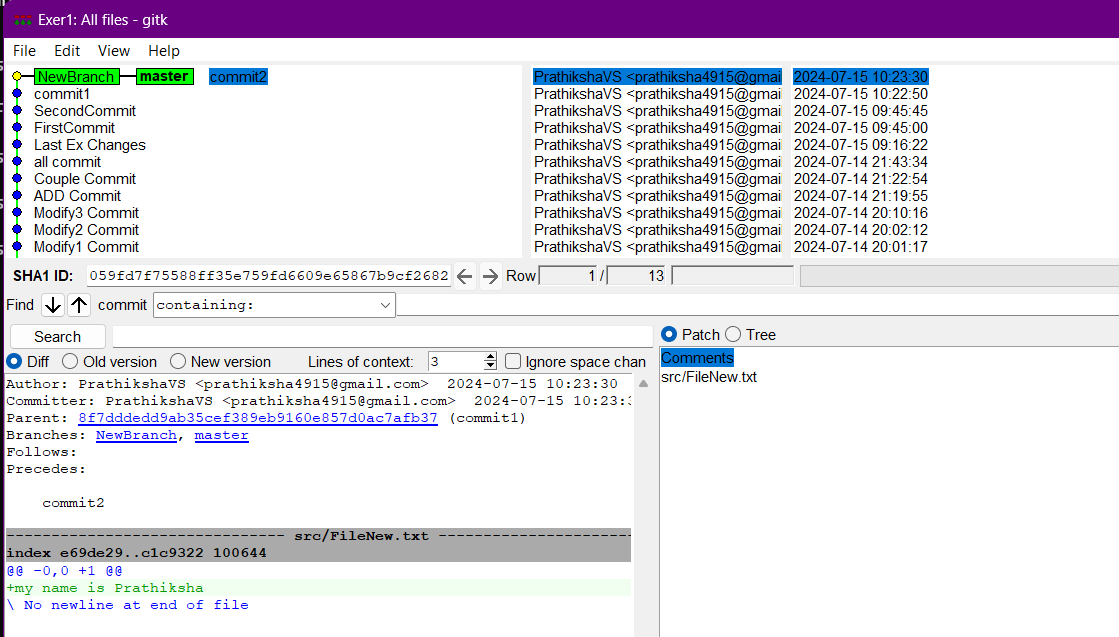
  
we cannot see the changes

1. Now merge your branch again. (Aside: you don’t need to do anything to inform Git that you only want to merge things added since your previous merge. Due to the way Git works, that kind of issue simply does not come up, unlike in early versions of Subversion



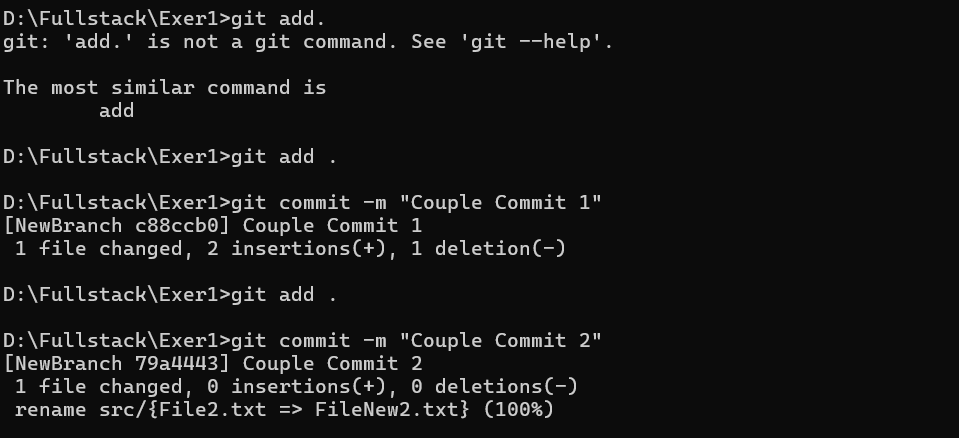
1. Look at git log. Notice that there is a merge commit. Also look in gitk. Notice the DAG now shows how things forked, and then were joined up again by a merge commit.



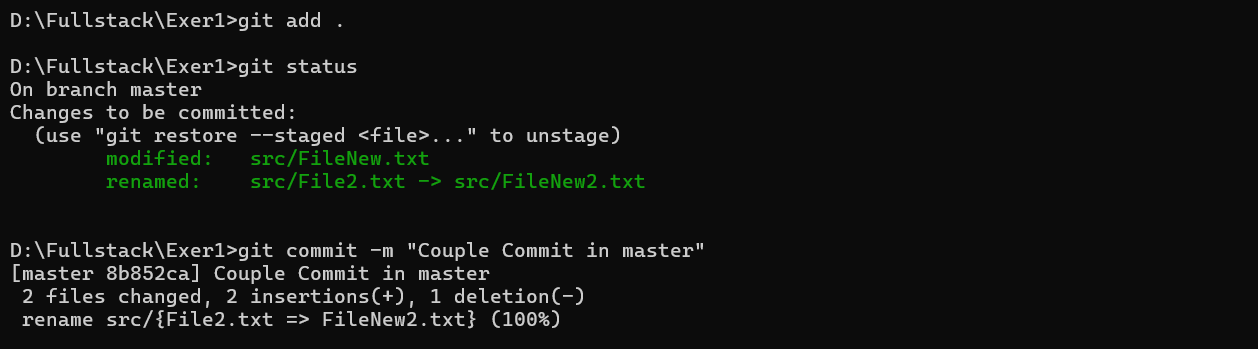


**Stretch Task**

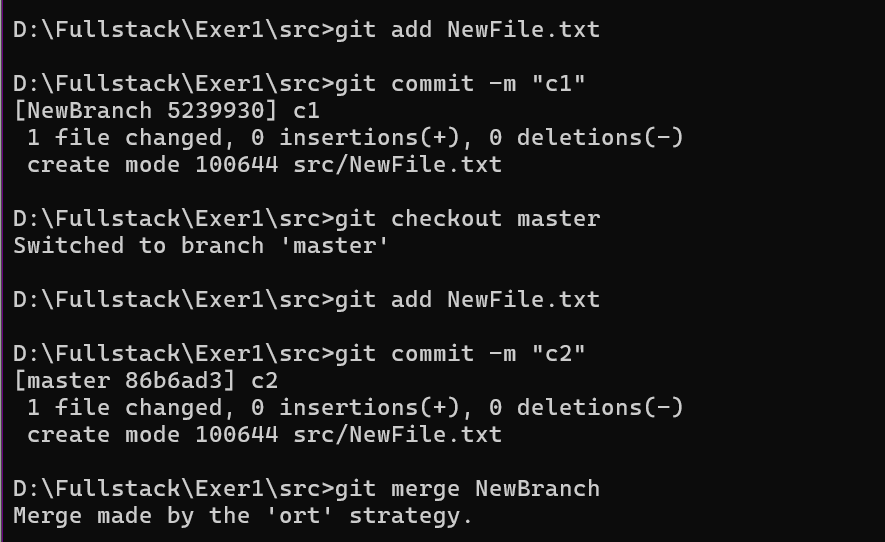
1. Once again, checkout your branch. Make a couple of commits.



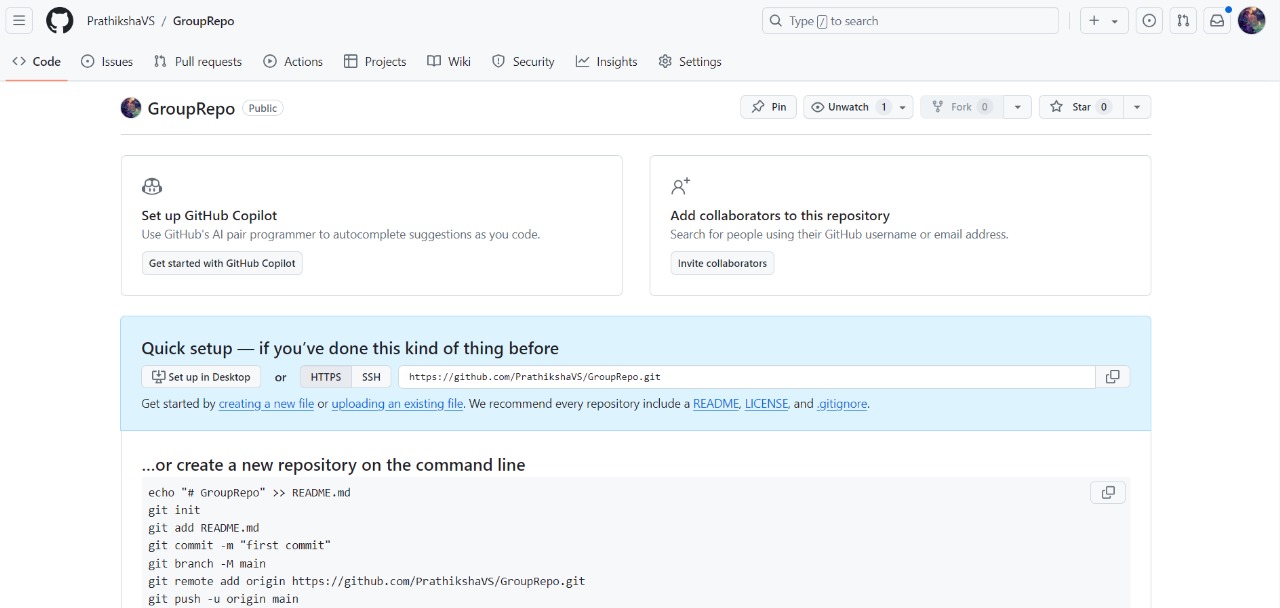
1. Return to your master branch. Make a commit there that changes the exact same line, or lines, as commits in your branch did.



1. Now try to merge your branch. You should get a conflict.



**Exercise 3**1. First, one person in the group should create a public repository using their GitHub account.



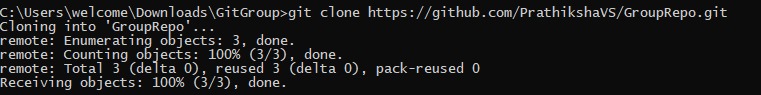
2. This same person should then follow the instructions from GitHub to add a remote, and then push their repository. Do not forget the –u flag, as suggested by GitHub!  
A screen shot of a computer

Description automatically generated

All of the other members of the group should then be added as collaborators, so they can commit to the repository also.  
A screenshot of a computer

Description automatically generated

4. Next, everyone else in the group should clone the repository from GitHub. Verify that the context of the repository is what is expected.  
A screenshot of a computer

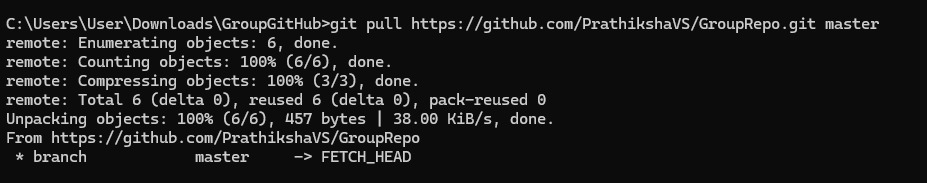
Description automatically generated  


A black screen with white text

Description automatically generated

5. One of the group members who just cloned should now make a local commit, then push it. Everyone should verify that when they pull, that commit is added to their local repository (use git log to check for it).  
A computer screen with a black square with white text

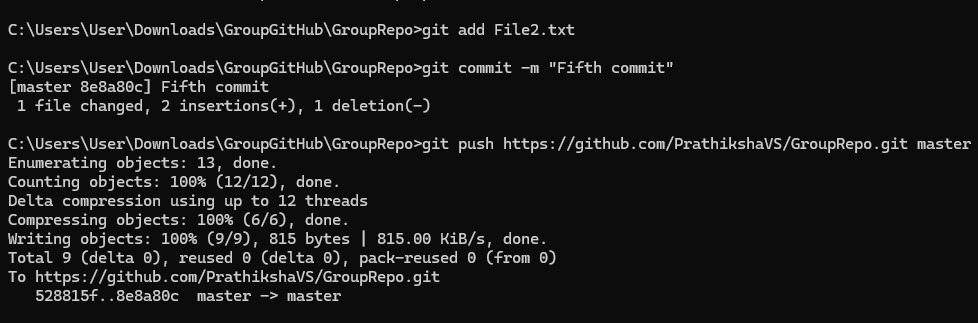
Description automatically generated  
A computer screen with white text

Description automatically generated  
 

6. Look at each other’s git log output. Notice how the SHA-1 is the same for a given commit across every copy of the repository. Why is this important?  
A computer screen with white text and numbers

Description automatically generated

7. Two members of the group should now make a **commit** locally, and race to **push** it. To keep things simple, be sure to edit different files. What happens to the runner-up?

  
A screen shot of a computer

Description automatically generated  
8. The runner-up should now pull. As a group, look at the output of the command. Additionally, look at the git log, and notice that there is a merge commit. You may also wish to view the DAG in gitk.

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A computer screen with a message

Description automatically generated  
9. Repeat the last two steps a couple of times, to practice.

A computer screen shot of a black screen

Description automatically generated

A computer screen shot of a black screen

Description automatically generated

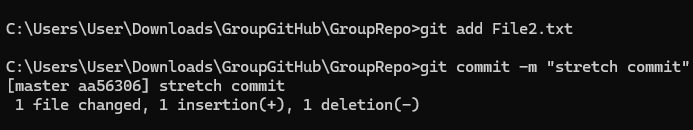
A screenshot of a computer program

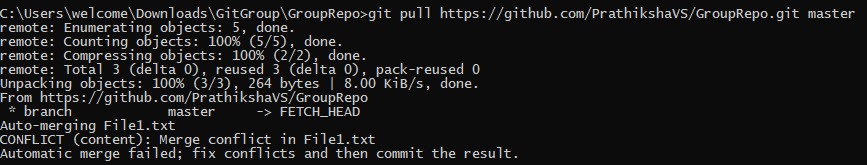
Description automatically generated

A screenshot of a computer

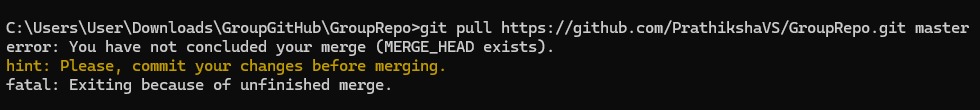
Description automatically generated  
**Stretch Task**

1. Now create a situation where two group members both edit the same line in the same file and **commit** it locally. Race to **push**.



  
2. When the runner-up does a pull, they should get a merge conflict  
A screenshot of a computer program

Description automatically generated

  
3. Look as a group at the file in conflict, and resolve it.  
  
A computer screen with white text

Description automatically generated

A computer screen with white text

Description automatically generated

4. Use the add command to stage the fix, and then use commit to make the merge commit. Notice how this procedure is exactly the one you got used to when resolving conflicts in branches.



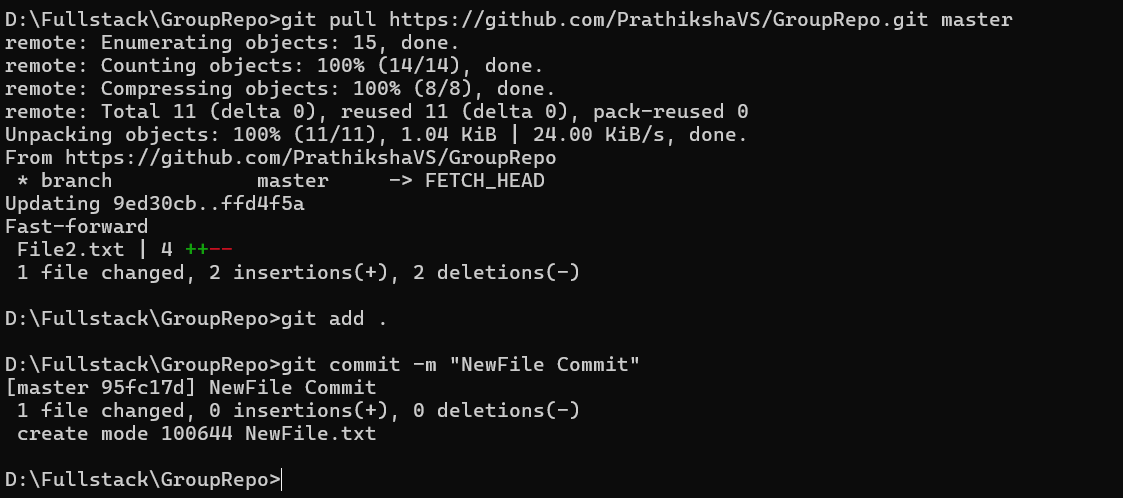
A computer screen with white text

Description automatically generated

**Exercise 4**

**Main Task**

1. Make a commit, and make a silly typo in the commit message



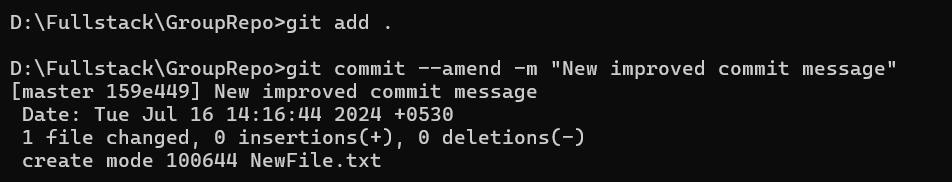
1. Use the --amend flag to enable you to fix the commit message.

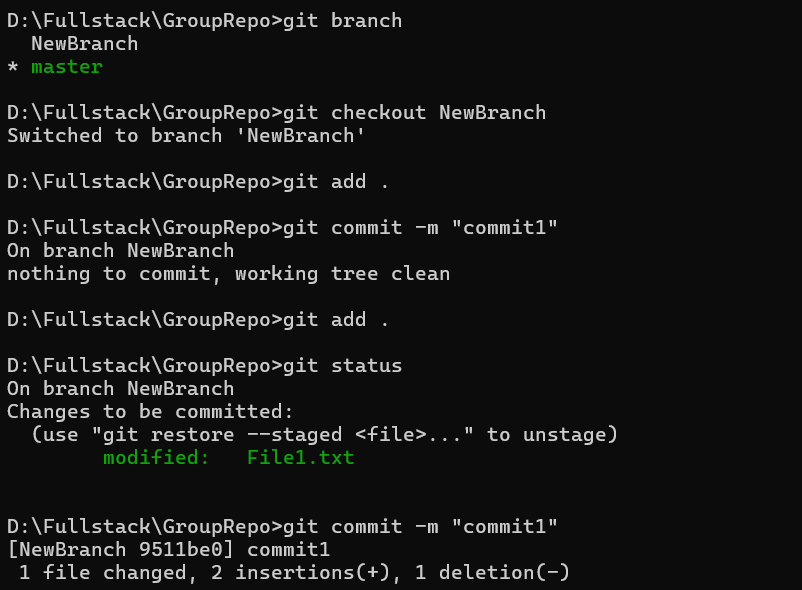


1. Look at the log and notice how the mistake is magically gone.

A screenshot of a computer screen

Description automatically generated

1. Now make a commit where you make a typo in one of the files. Once again, use --amend to magic away your problems.  
   
2. Create a branch. Make a commit.



1. Now switch back to your master branch. Make a (non-conflicting) commit there also.

A computer screen with white text

Description automatically generated

1. Now switch back to your branch.

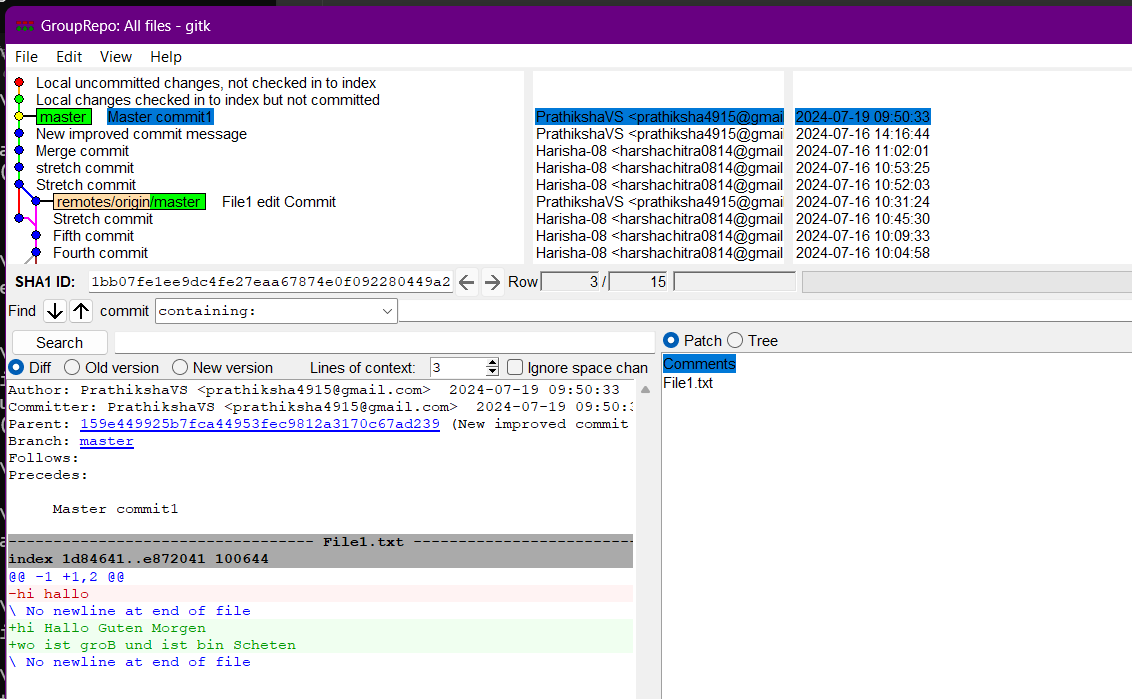
A screen shot of a computer program

Description automatically generated

1. Use the rebase command in your branch. Look at the DAG in gitk, and note that you have the commit from the master branch, but no merge commit.

A screenshot of a computer program

Description automatically generated



1. Make one more commit in your branch.

A black screen with white text

Description automatically generated

1. Return to master. Merge your branch. Notice how, thanks to the rebase, this is a fast-forward merge.

A black background with white text

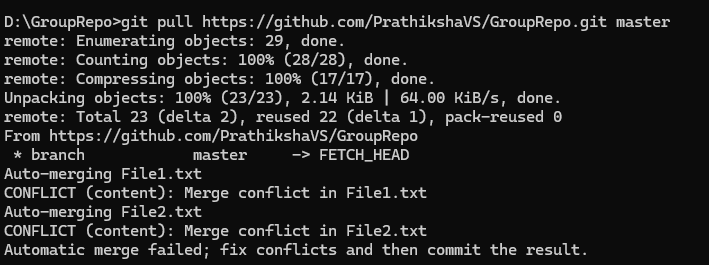
Description automatically generatedA black screen with white text

Description automatically generated

**Stretch Task**

1. Find somebody from your team from the previous exercise. Have them **push** a commit to the central repository. A computer screen with white text

   Description automatically generated
2. Make a commit locally yourself also. Note that you should not have pulled their commit at this point.



1. Try to **push**, and watch it fail.

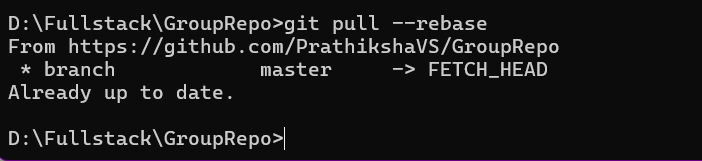
A screen shot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

1. Now, **pull** but using the **--rebase** flag.



1. Use **git log** and **gitk** to verify that there is no merge commit, and the DAG is linear.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

1. Notice that your commit is the latest one, even though temporally the other member of your team made their commit afterwards. Why is this?   
   This is because of the git pull - - rebase command. When we ran Git rewrote your local commit to be on top of the latest commit from the remote repository.