# Naveenkumar M

# 73772126135

# B.Tech AI&DS

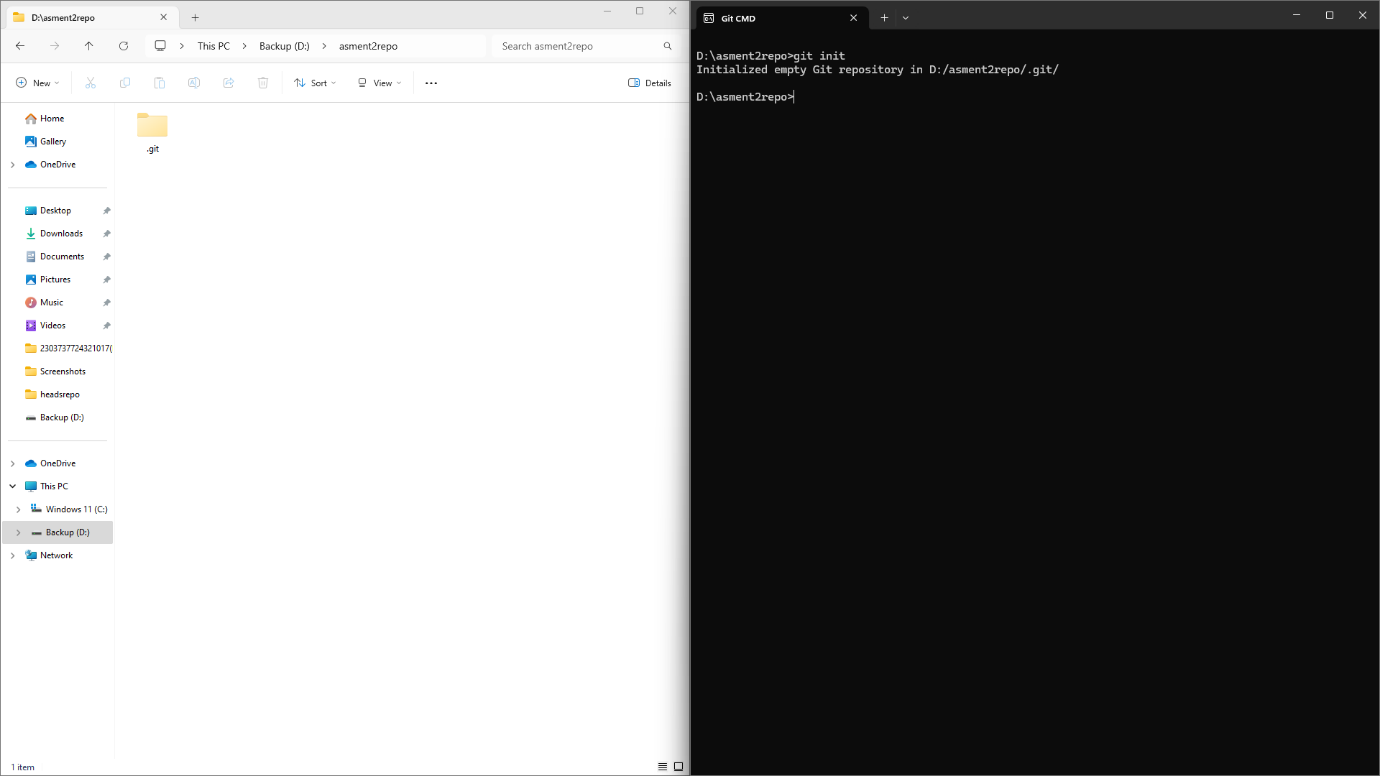
# Email:naveenkumarksrct35@gmail.com

# 13.7.2024

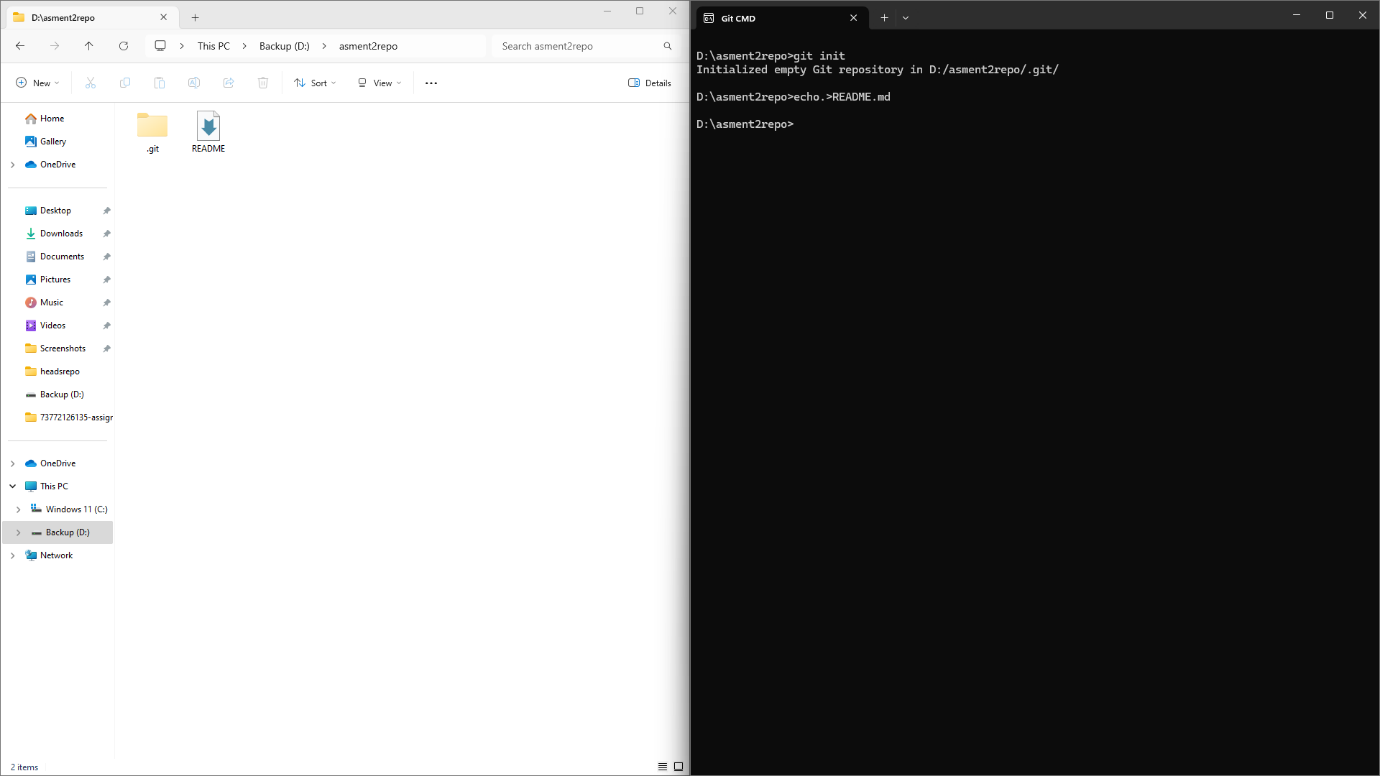
# Assignment-2

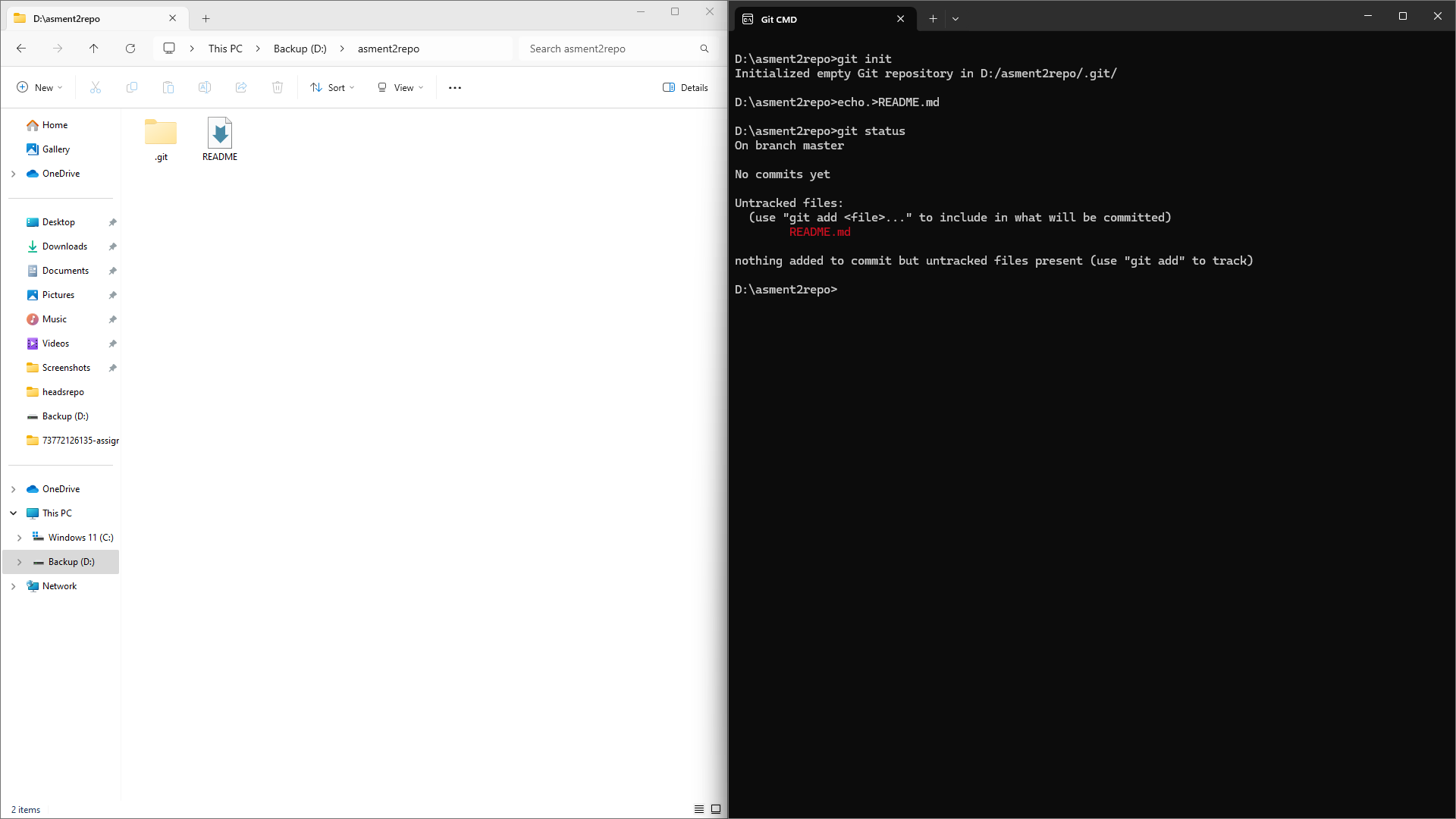
Exercise -1

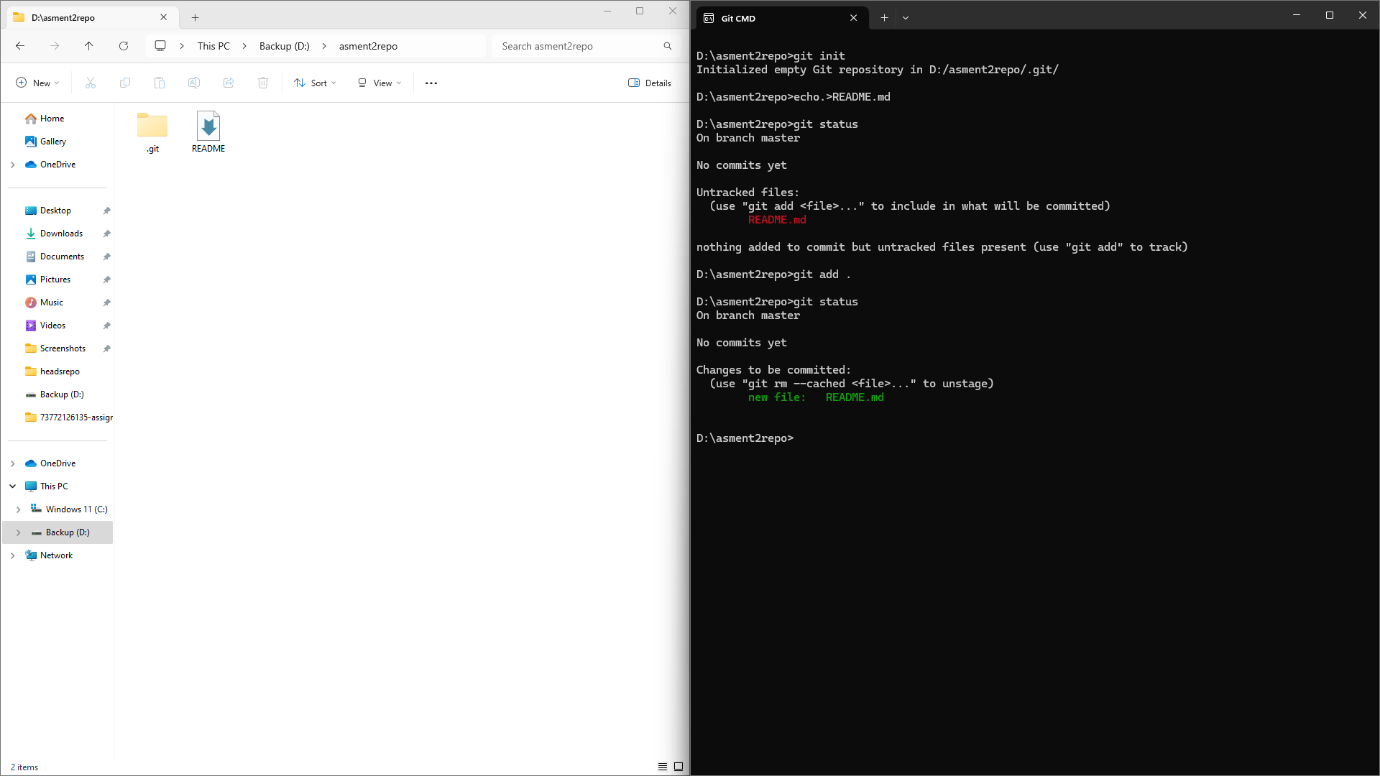
1. Use the init command to create a Git repository in that directory.



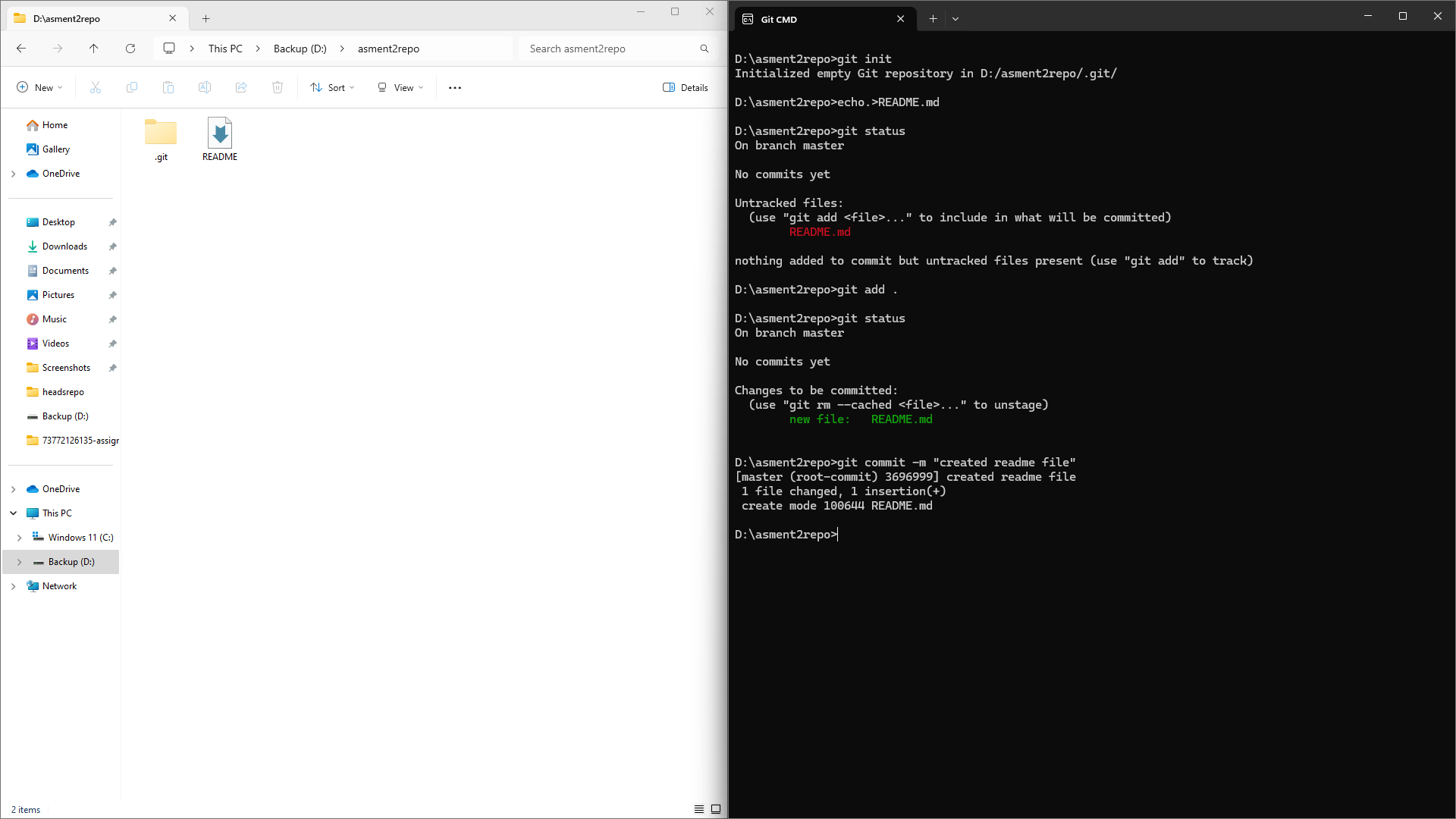
2.Create a README file.



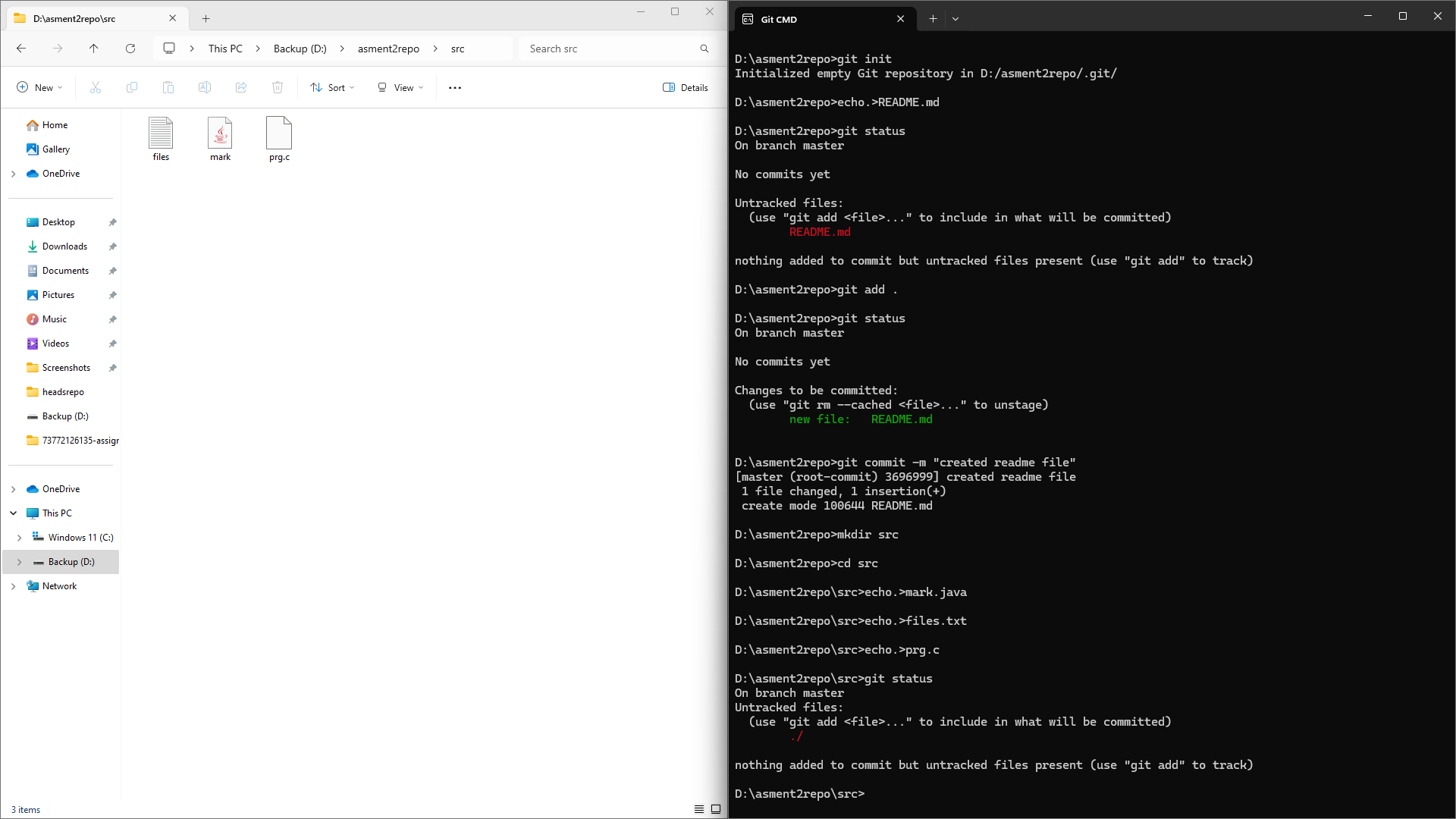
3.Look at the output of the status command; the README you created should appear as an untracked file.

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

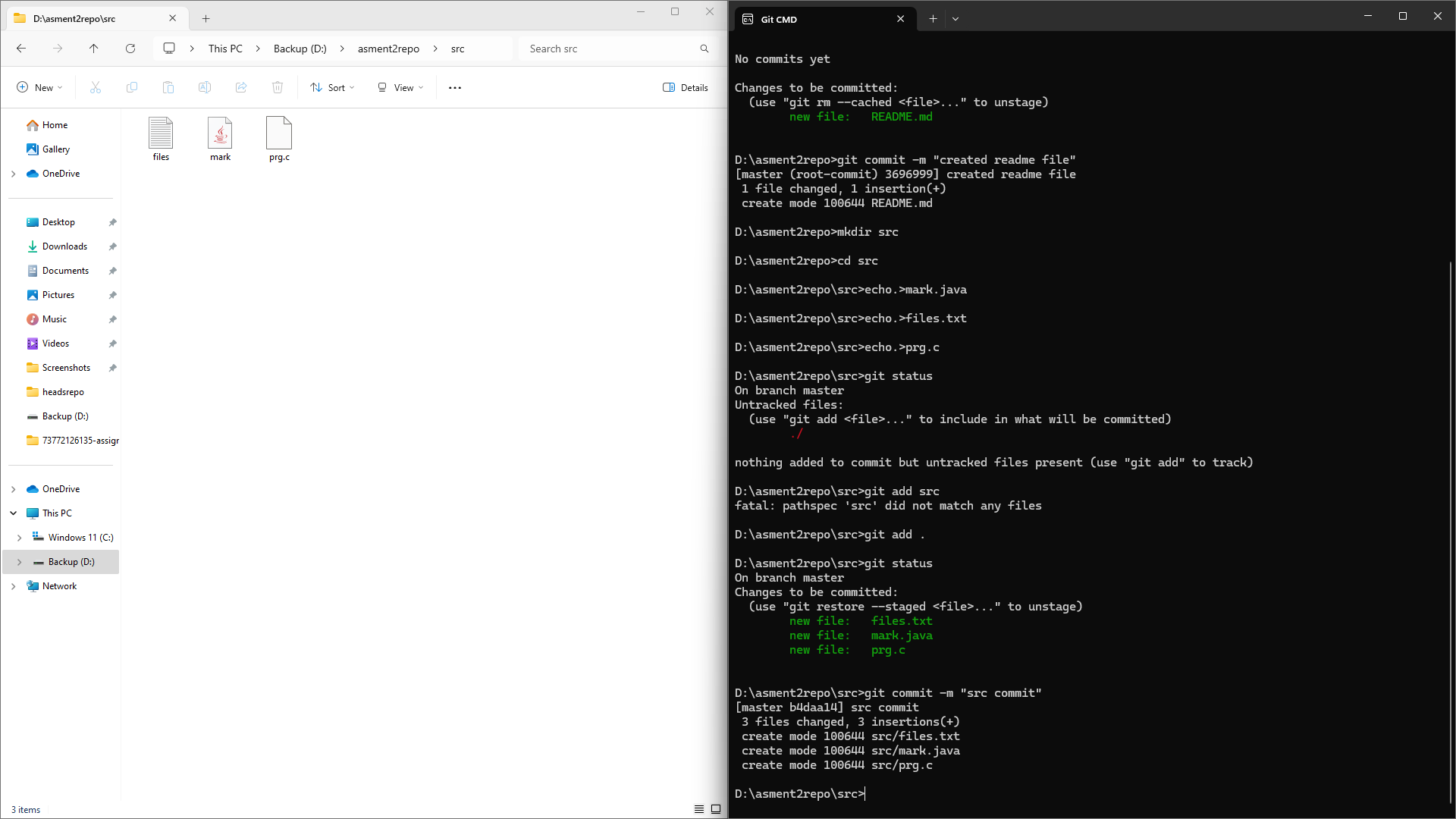
5.Now use the commit command to commit the contents of the staging area.



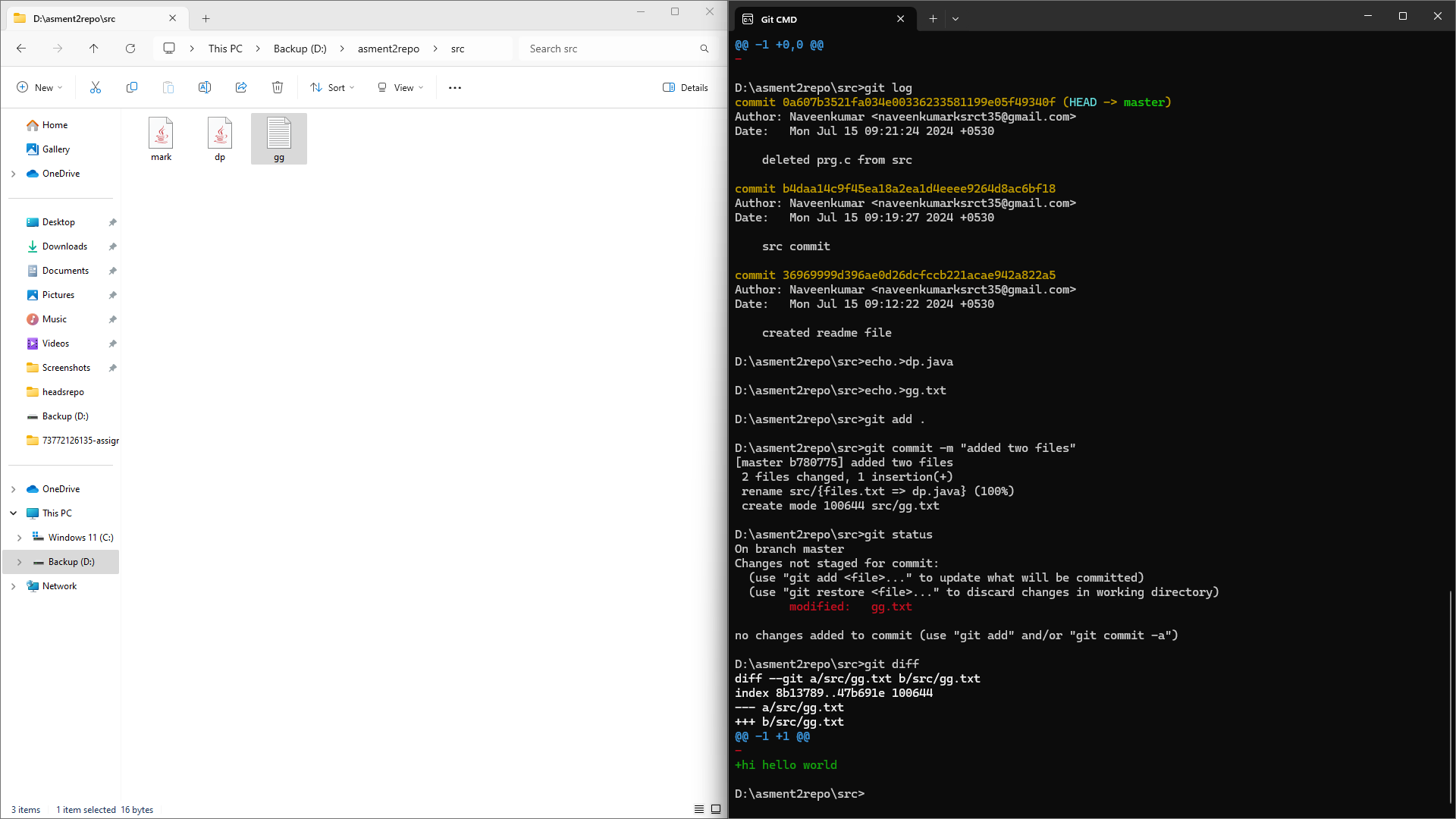
6.Create a src directory and add a couple of files to it.



7.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.



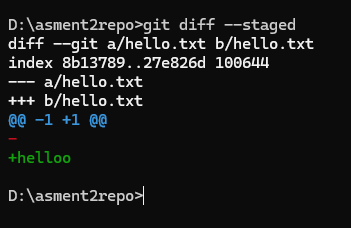
8.Make a change to one of the files. Use the diff command to view the details of the change.

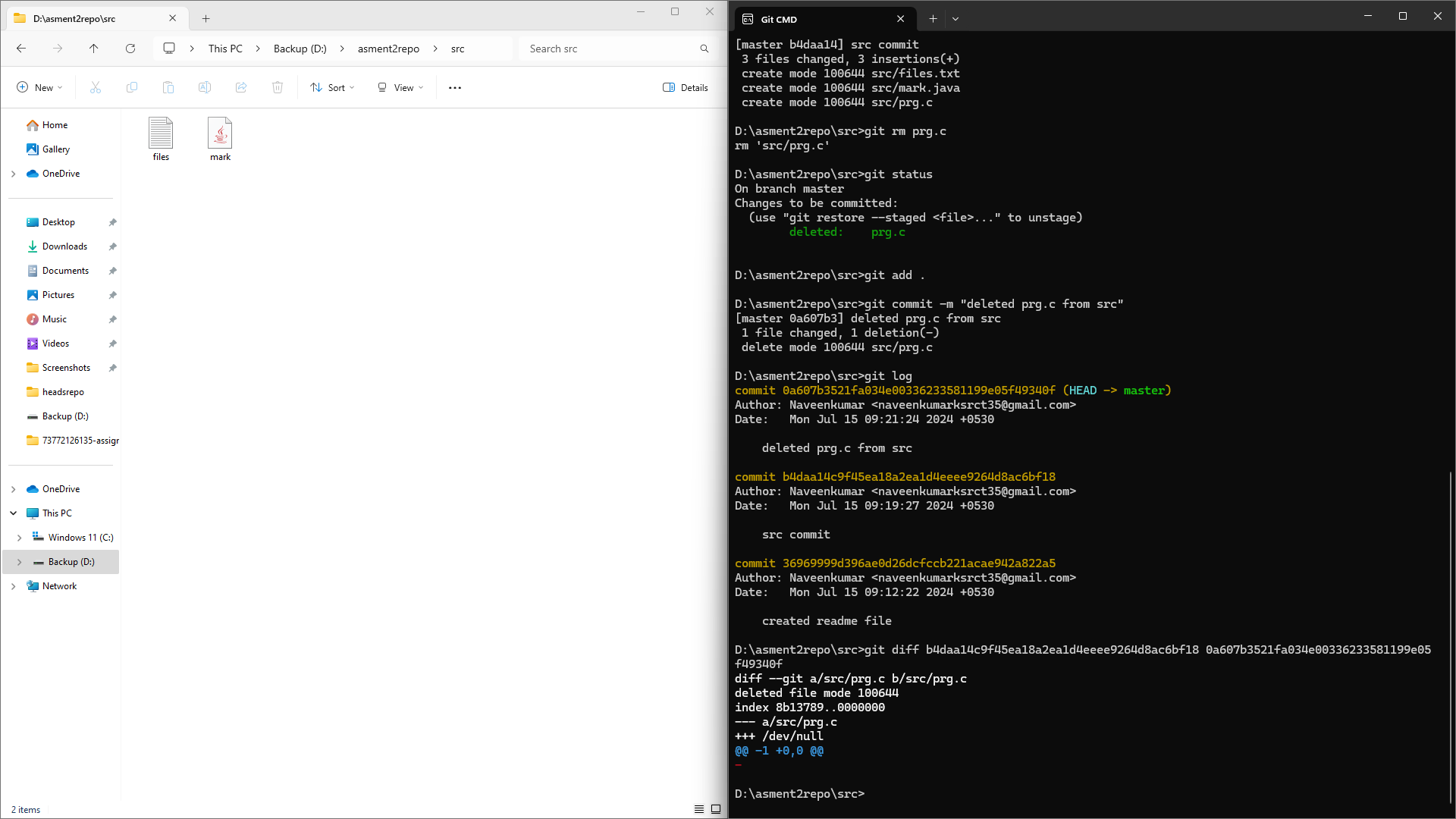


9.Next, add the changed file, and notice how it moves to the staging area in the status output. Also observe that the diff command you did before using add now gives no output. Why not? What do you have to do to see a diff of the things in the staging area? (Hint: review the slides if you can’t remember.)

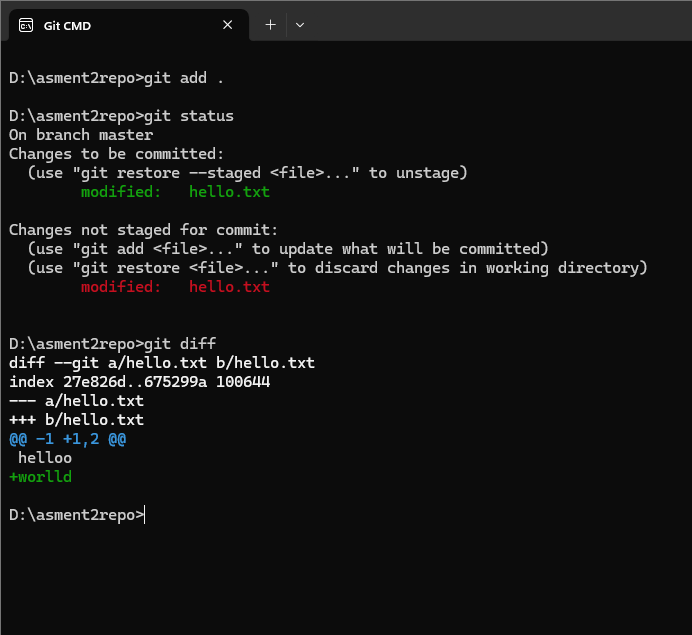
Ans:

The git diff shows nothing because the changes has been staged so git diff doesn’t provide diff of staged changes but you can use git diff --staged to see the changes.

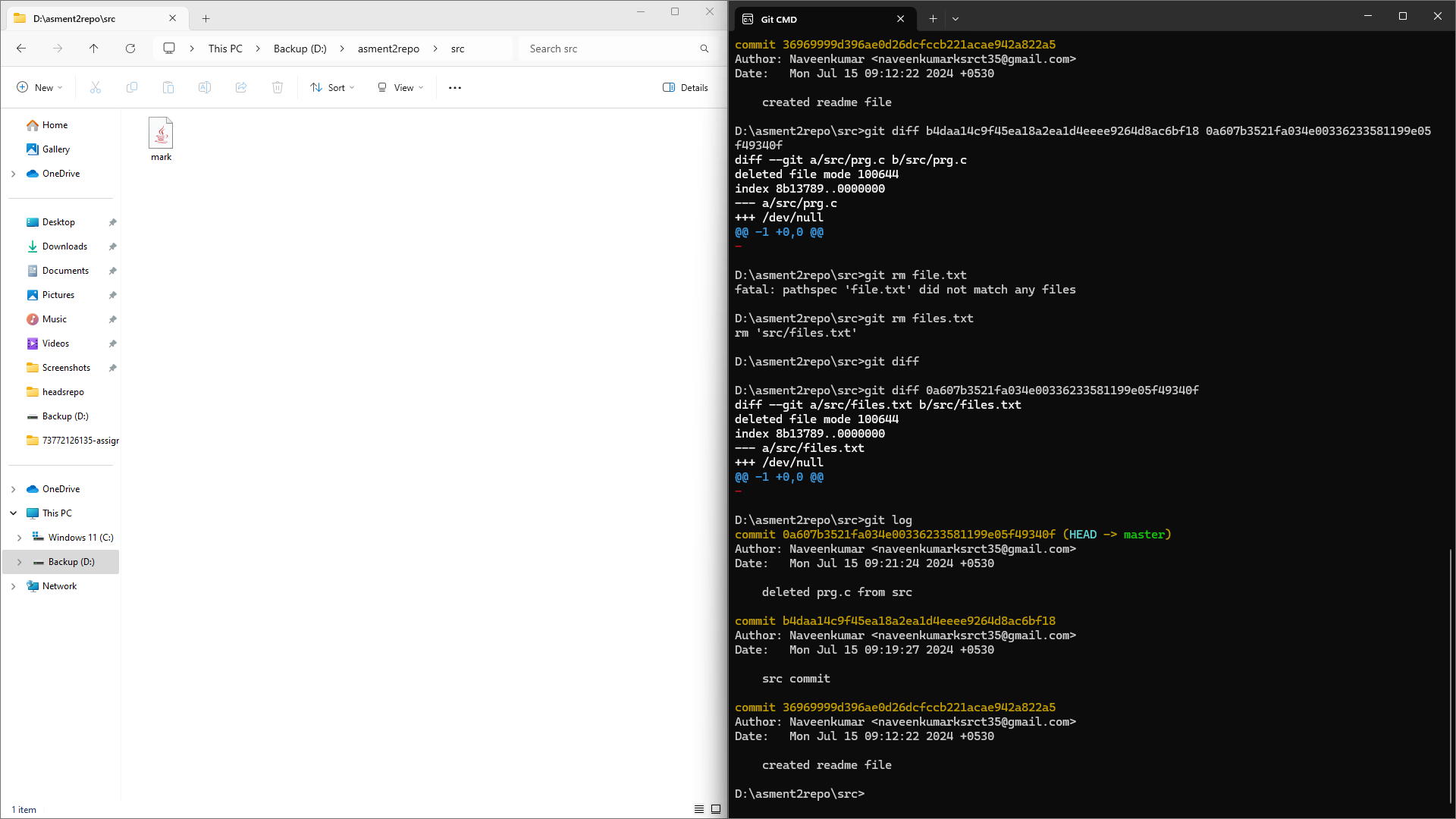




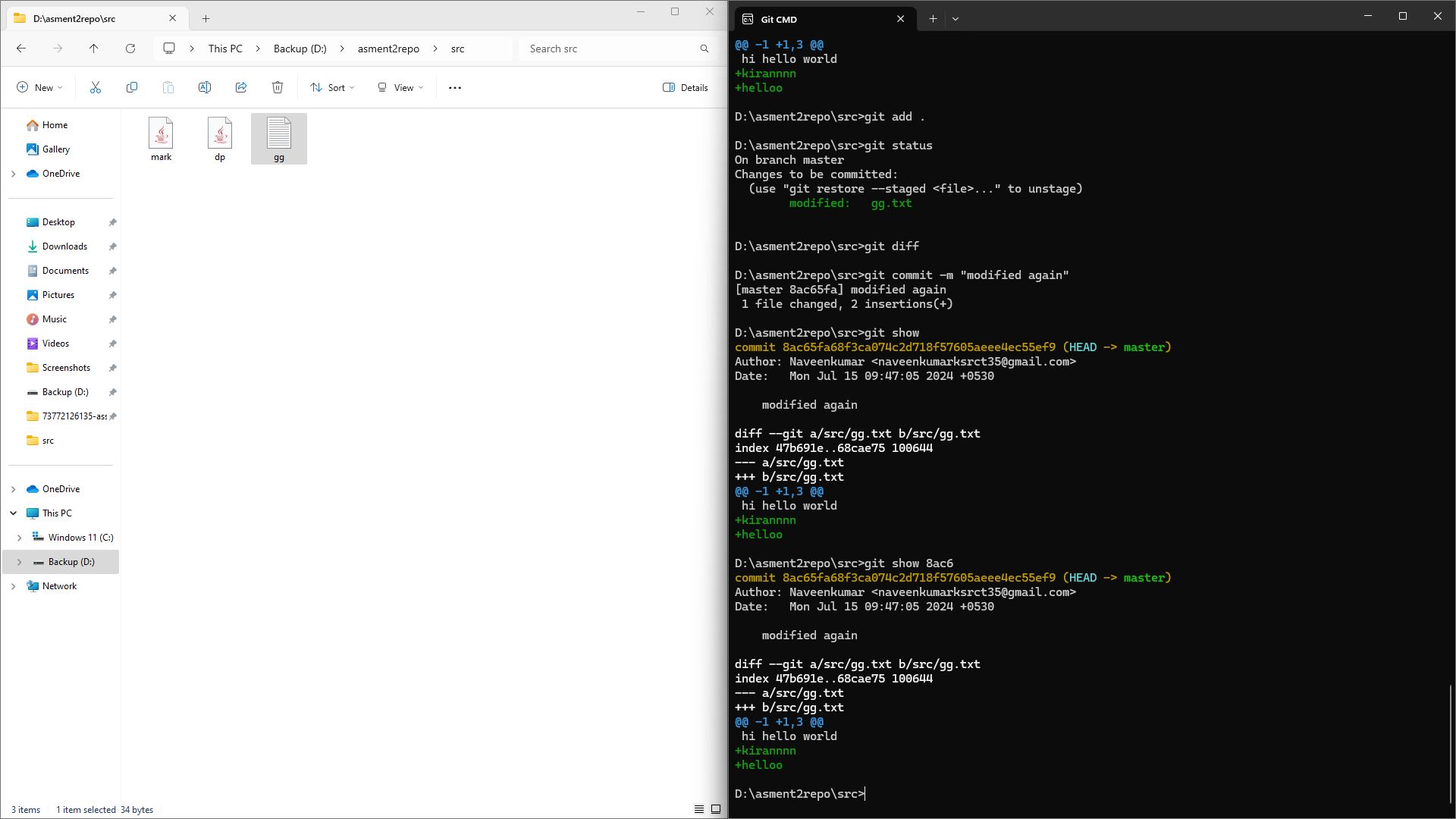
10.Now – without committing – make another change to the same file you changed in step 10. Look at the status output, and the diff output. Notice how you can have both staged and unstaged changes, even when you’re talking about a single file. Observe the difference when you use the add command to stage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



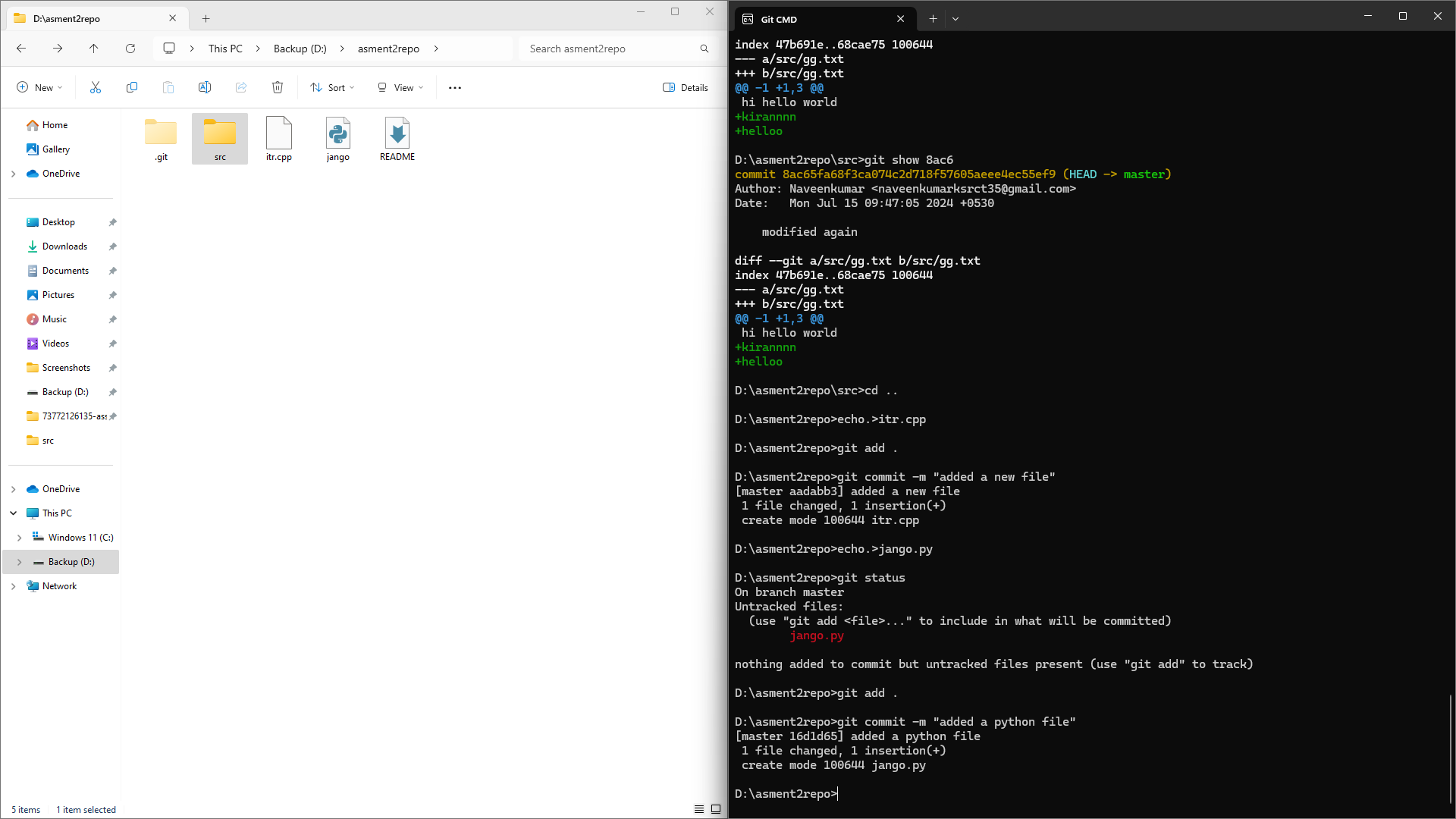
11.Use the log command in order to see all of the commits you made so far.



12. 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?

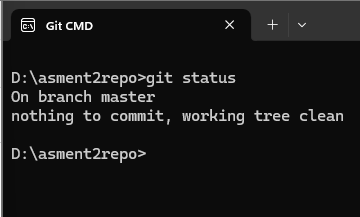


13.Make a couple more commits, at least one of which should add an extra file.



Exercise -2

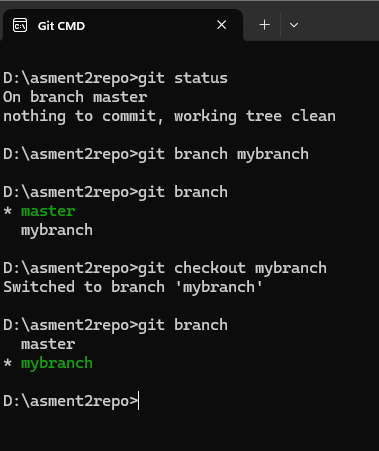
1.Run the status command. Notice how it tells you what branch you are in.



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

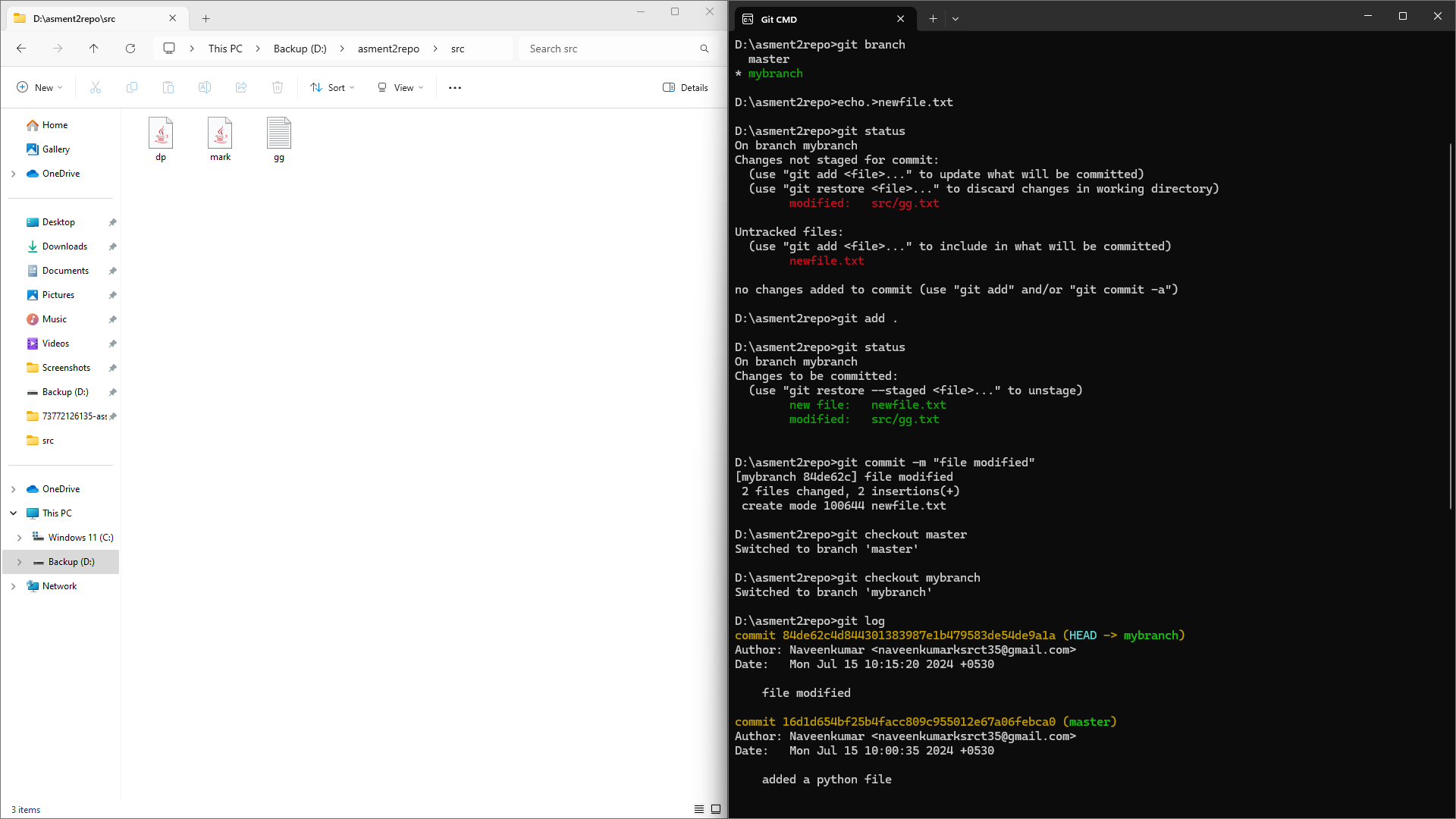


3. Use the checkout command to switch to it.

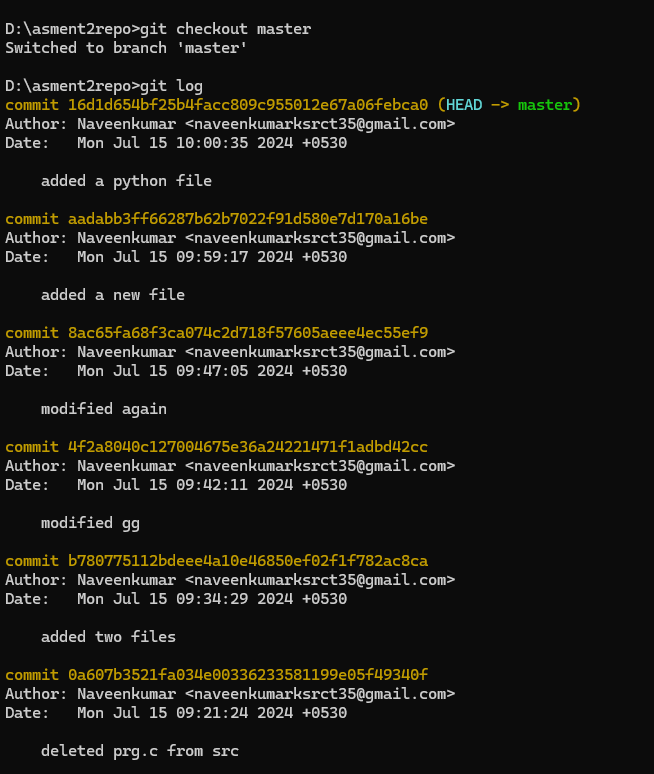


4. Make a couple of commits in the branch – perhaps adding a new file and/or editing existing ones.

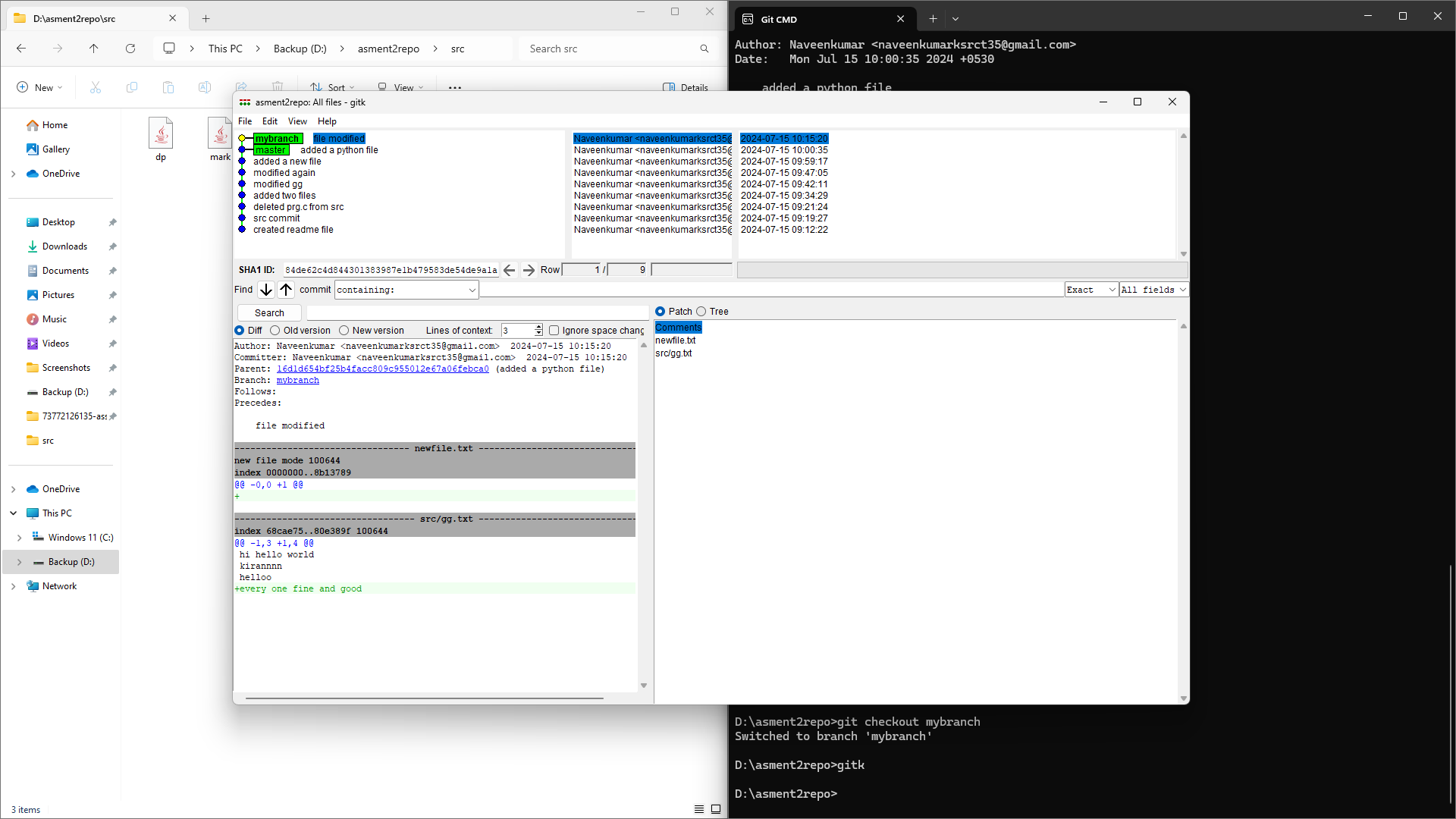
5. Use the log command to see the latest commits. The two you just made should be at the top of the list.



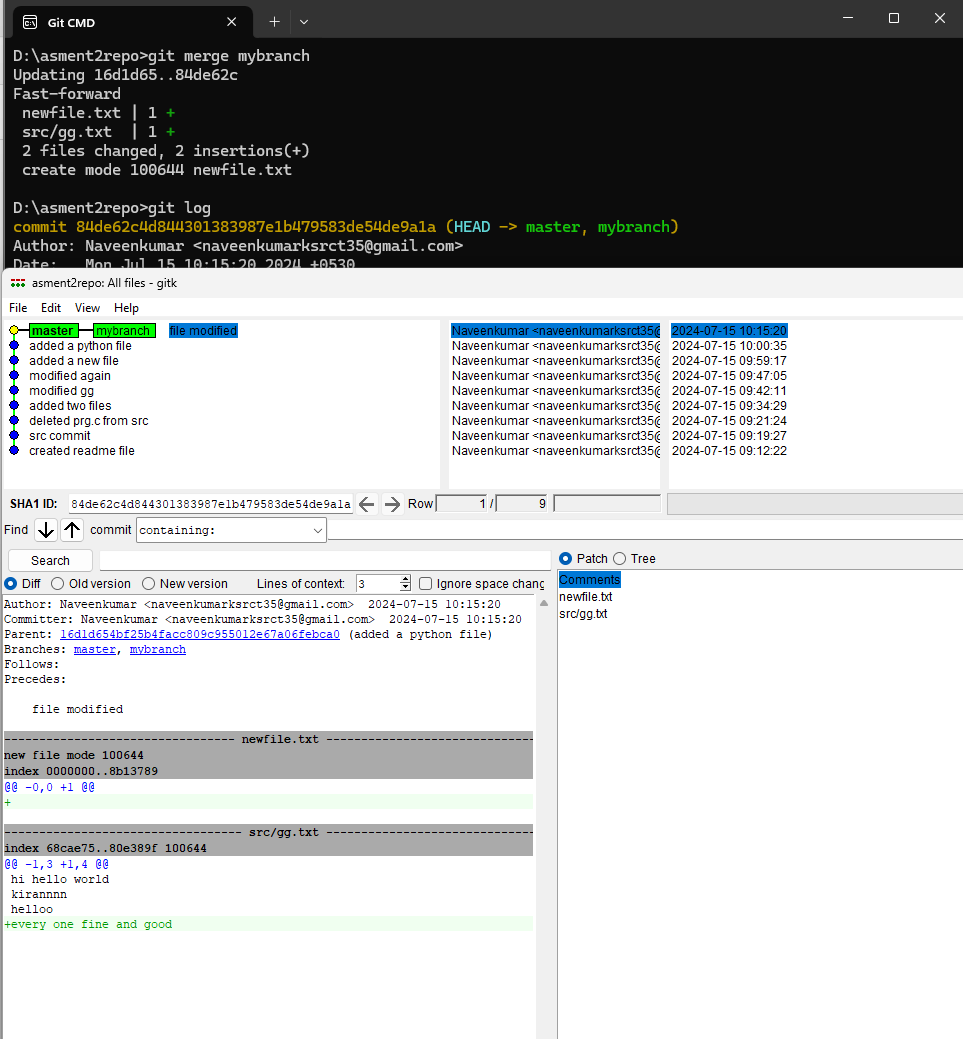
6. Use the checkout command to switch back to the master branch. Run log again. Notice your commits don’t show up now. Check the files also – they should have their original contents.



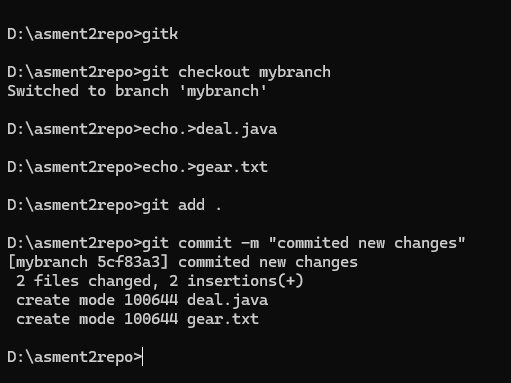
7. Use the checkout command to switch back to your branch. Use gitk to take a look at the commit graph; notice it’s linear.



8. 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.

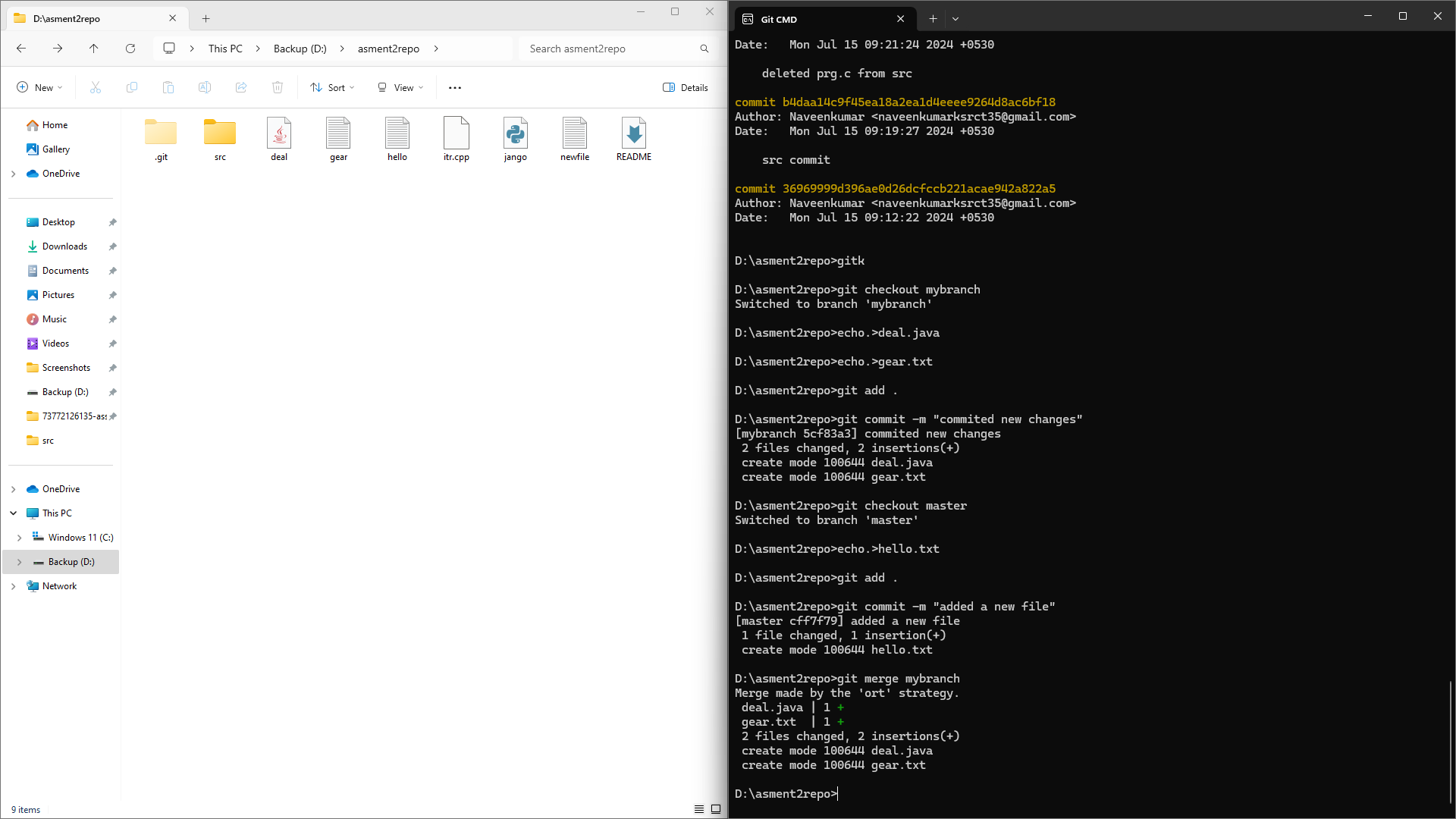


9. Switch back to your branch. Make a couple more commits.

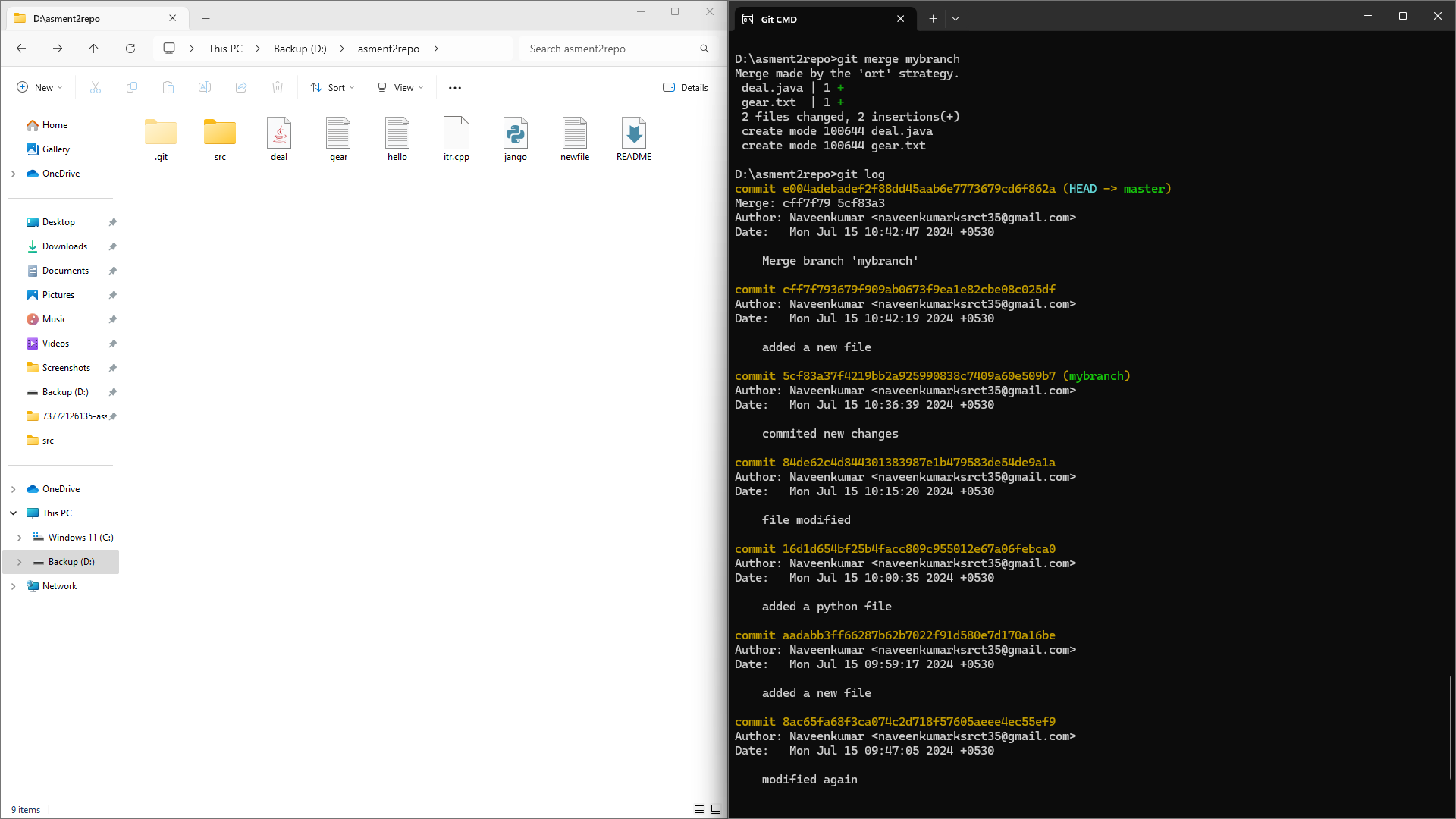


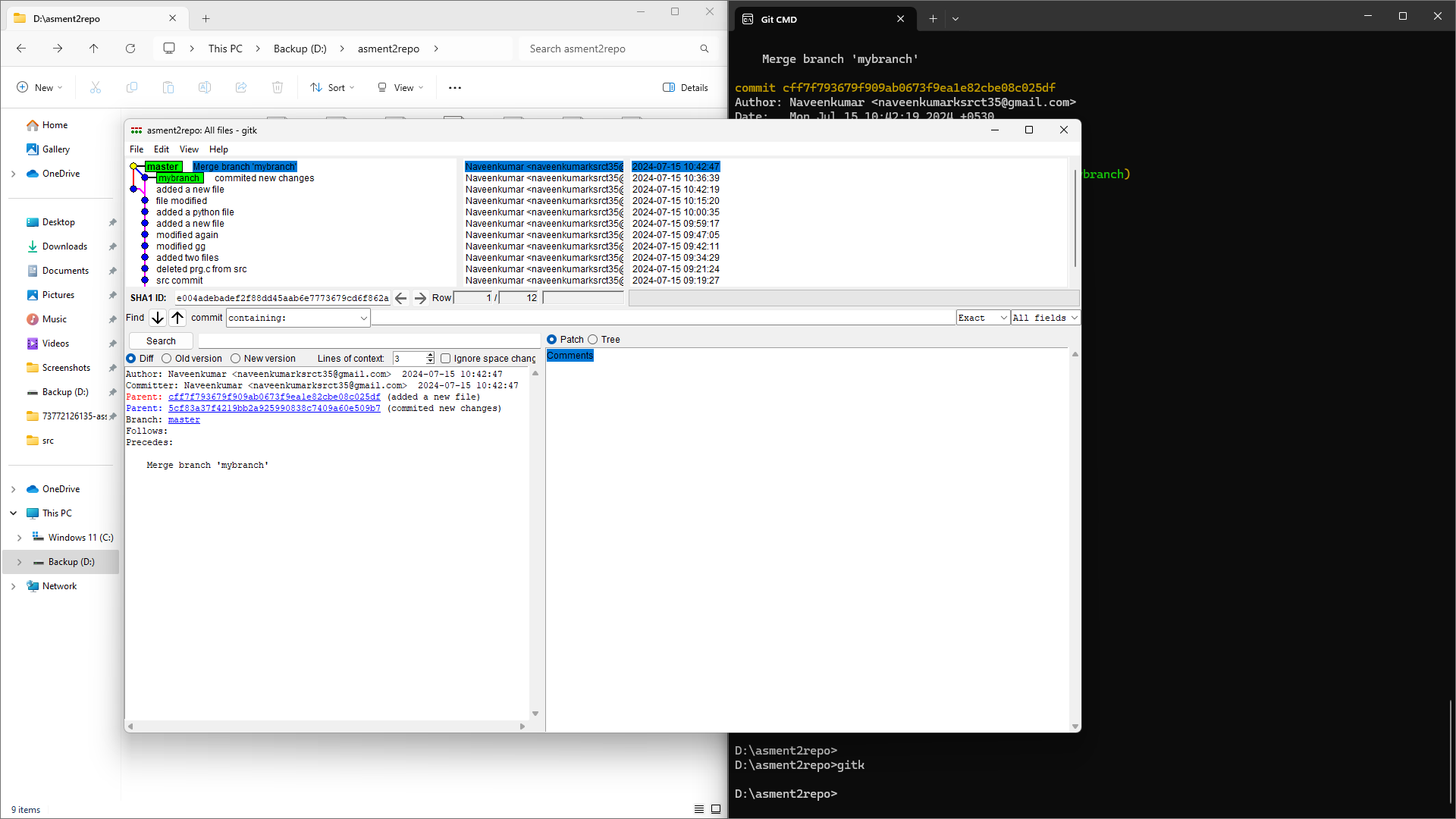
10. Switch back to master. Make a commit there, which should edit a different file from the ones you touched in your branch – to be sure there is no conflict.

11. 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.)



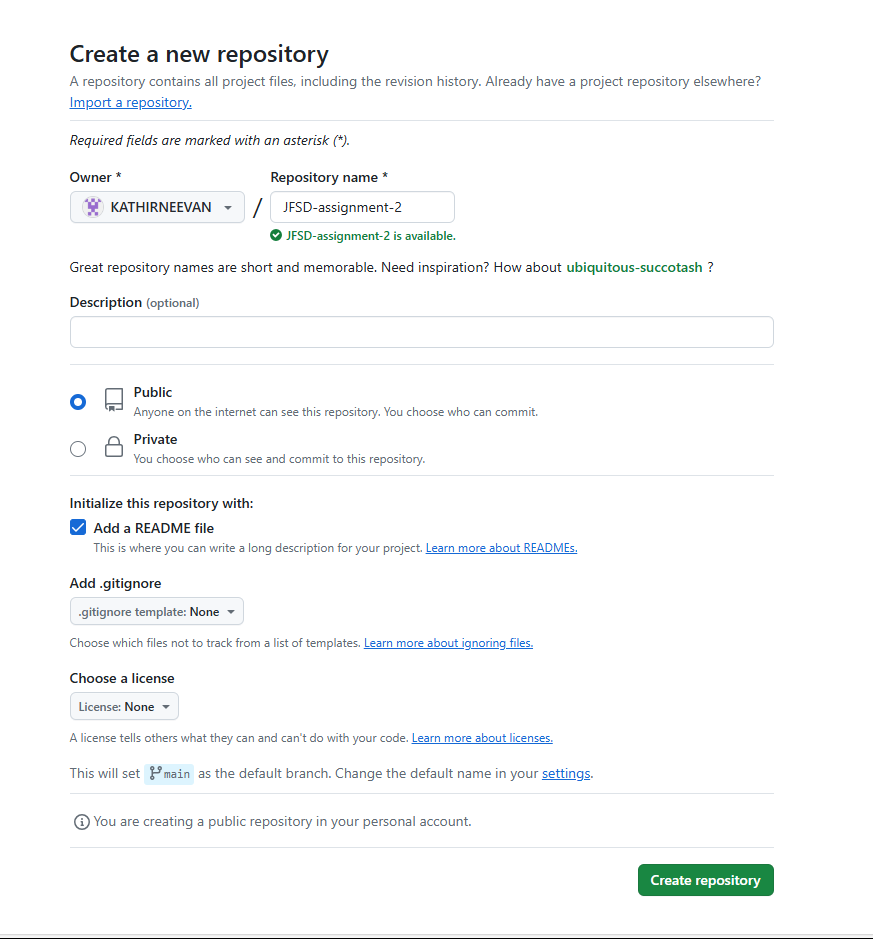
12. Look at git log. Notice that there is a merge commit. Also look in gitk. Notice the DAG now shows how things forked, and then were joined up again by a merge commit.

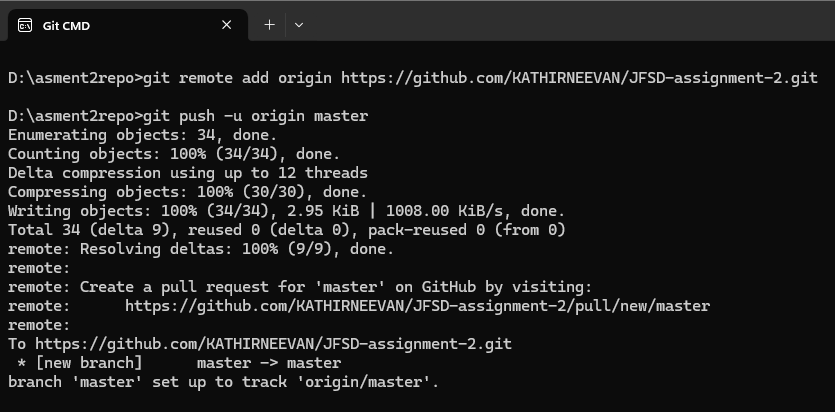




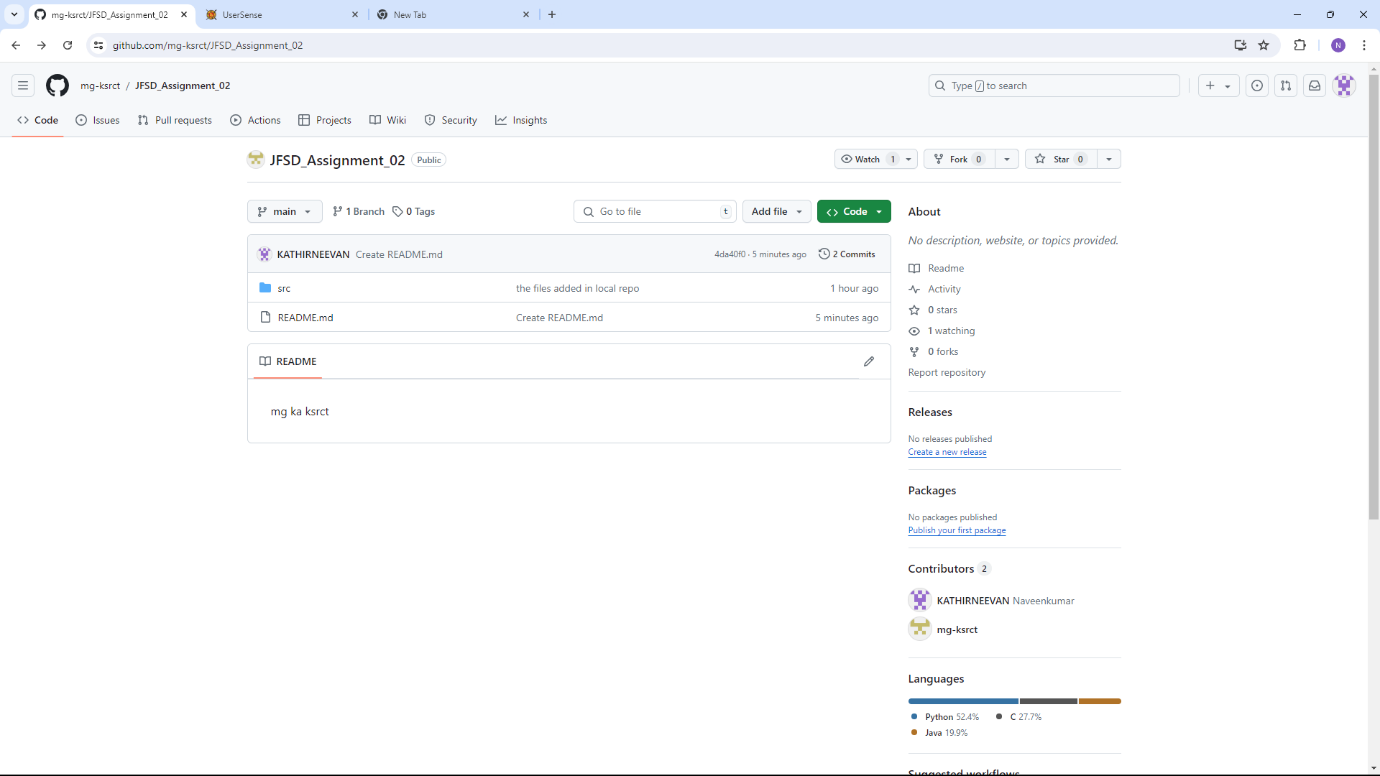
Exercise-3

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

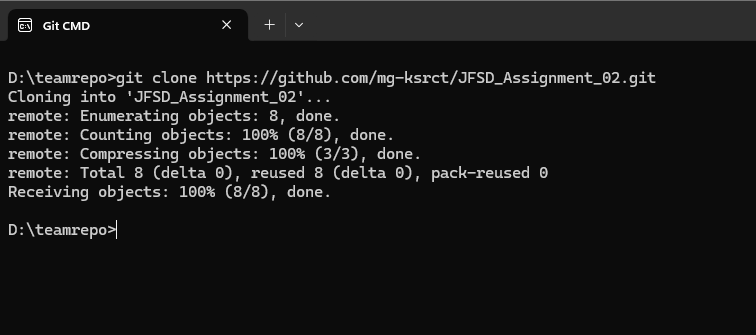


2. This same person should then follow the instructions from GitHub to add a remote, and then push their repository. Do not forget the –u flag, as suggested by GitHub!

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

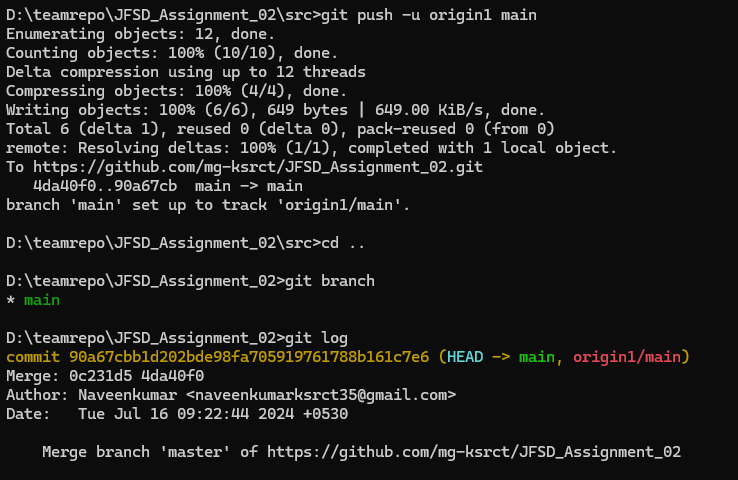


4. Next, everyone else in the group should clone the repository from GitHub. Verify that the context of the repository is what is expected.

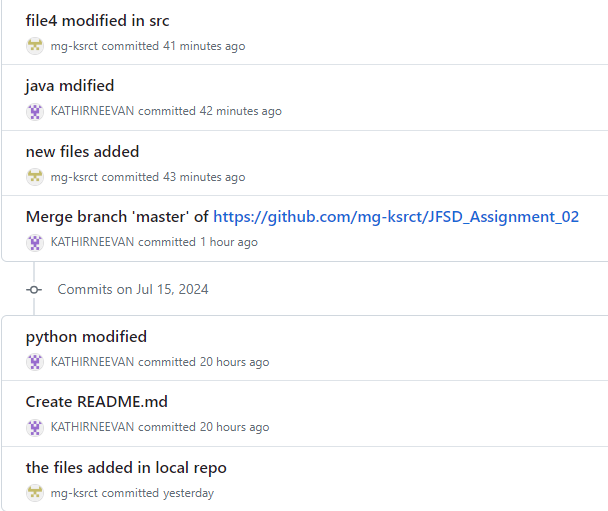


5. One of the group members who just cloned should now make a local commit, then push it. Everyone should verify that when they pull, that commit is added to their local repository (use git log to check for it).

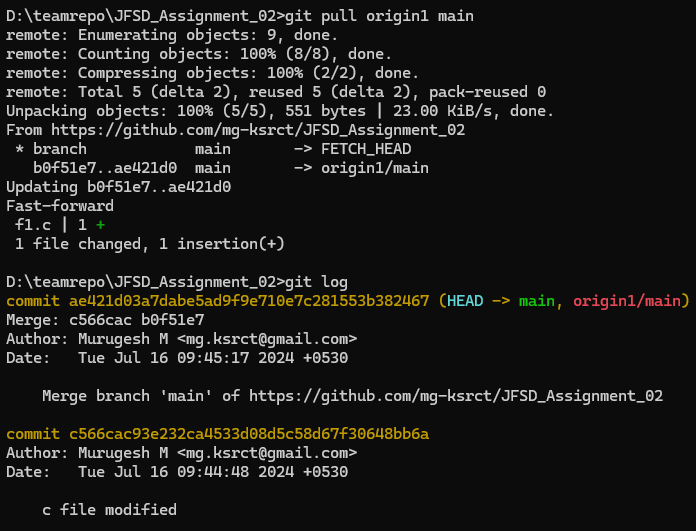
6. Look at each other’s git log output. Notice how the SHA-1 is the same for a given commit across every copy of the repository. Why is this important?



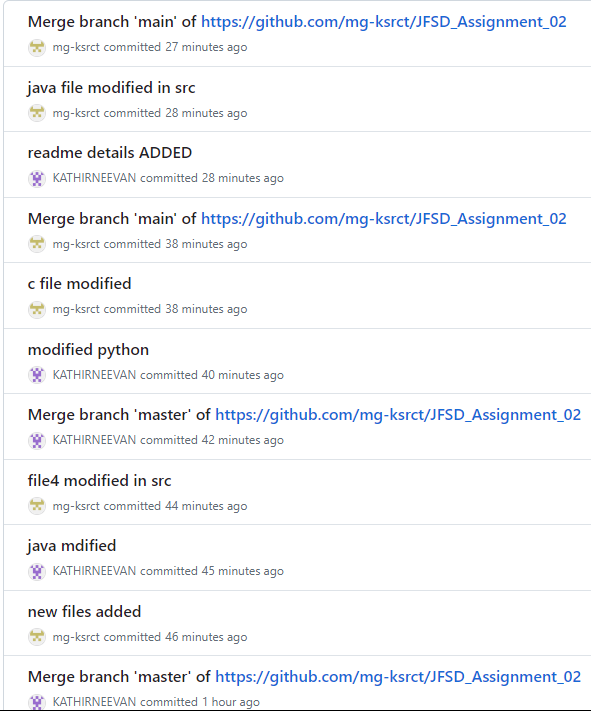
7. Two members of the group should now make a commit locally, and race to push it. To keep things simple, be sure to edit different files. What happens to the runner-up?

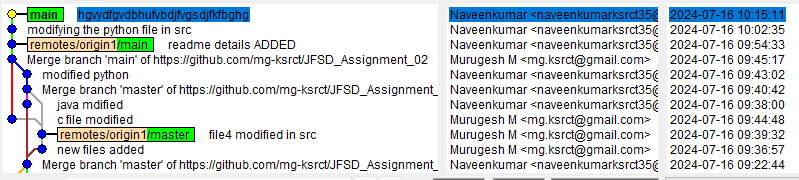


8. The runner-up should now pull. As a group, look at the output of the command. Additionally, look at the git log, and notice that there is a merge commit. You may also wish to view the DAG in gitk



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



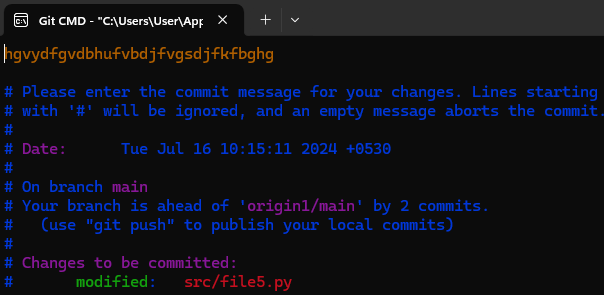


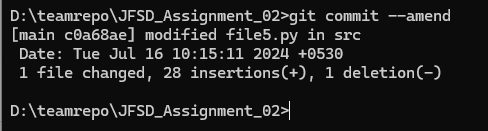
EXERCISE-4

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

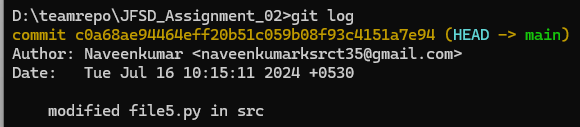


2. Use the --amend flag to enable you to fix the commit message.

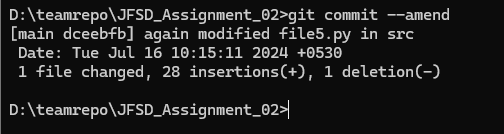


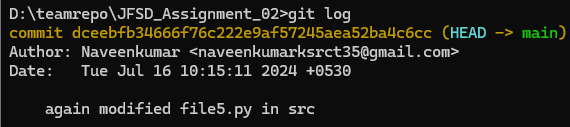


3. Look at the log and notice how the mistake is magically gone.



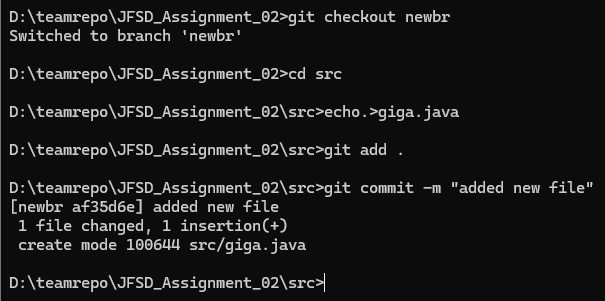
4. Now make a commit where you make a typo in one of the files. Once again, use --amend to magic away your problems.



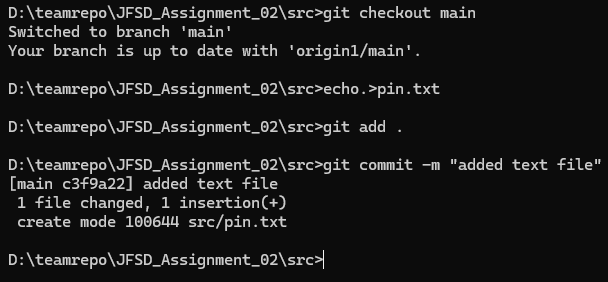


5. Create a branch. Make a commit.

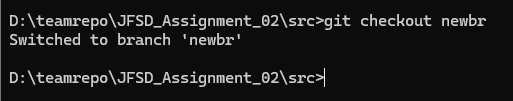




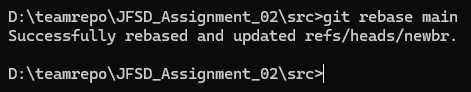
6. Now switch back to your master branch. Make a (non-conflicting) commit there also.

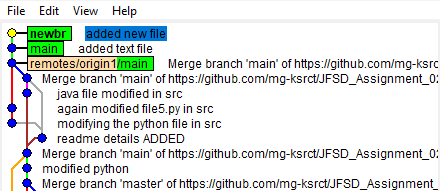


7. Now switch back to your branch.



8. 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.

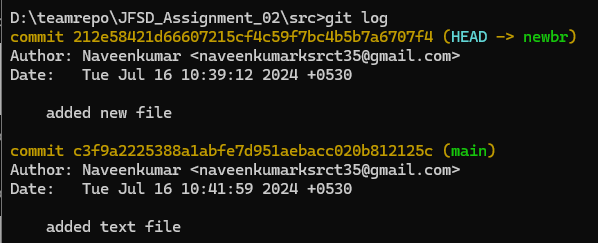




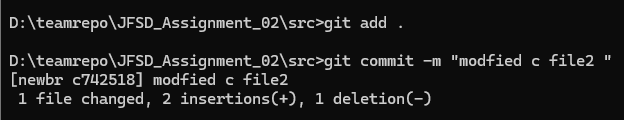
Note: Yeah there is no merge commit because this rebase command only gives the new changes to our branch that do not merge branch

My last change or commit in main branch is added to my current newbr branch.

This reflects in my git log also



9. Make one more commit in your branch.



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

