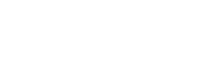
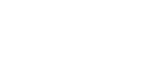


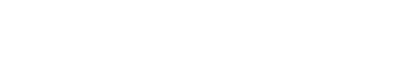
Subject Name:



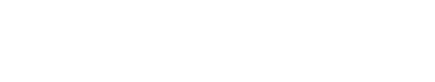
**Source**



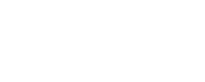
**Code**



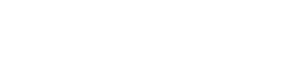
**Management**



Subject Code:



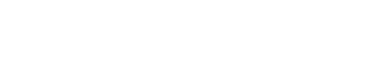
**CS181**



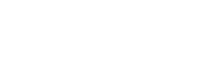
Cluster: Z



**eta**



Department:



**DCSE**



**Submitted By:**  **Submitted To:**

PRINCE GUPTA **Type your text****Type your text**Dr. Anuj Jain

2110992024

# G-27



# **List of Tasks**

|  |  |  |
| --- | --- | --- |
| **S.**  **No** | **Task Title** | **Page No.** |
| 1 | Add collaborators on GitHub Repository |  |
| 2 | Fork and Commit |  |
| 3 | Merge and Resolve conflicts created due to own activity and collaborators activity |  |
| 4 | Reset and Revert |  |

**Task:1**

**Inviting collaborators to a personal repository**

You can invite users to become collaborators to your personal repository.

If you're using GitHub Free, you can add unlimited collaborators on public and private repositories.

Repositories owned by an organization can grant more granular access. For more information, see "[Access permissions on GitHub."](https://docs.github.com/en/articles/access-permissions-on-github)

Pending invitations will expire after 7 days, restoring any unclaimed licenses.

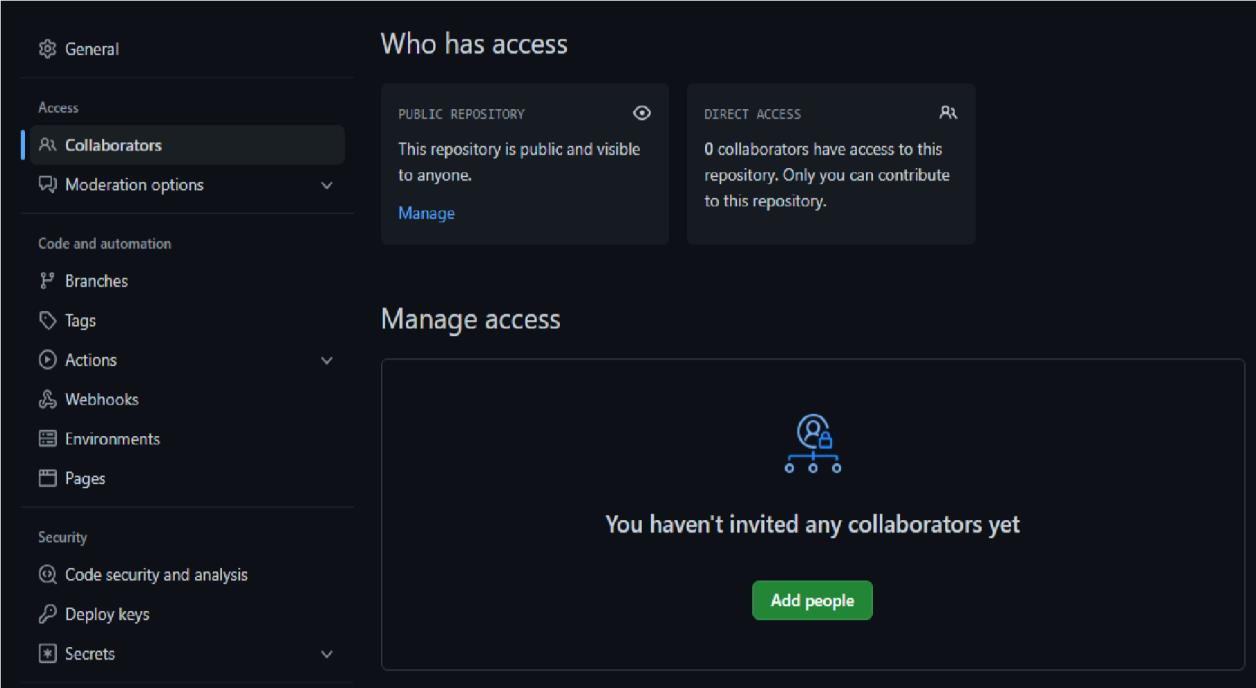
If you're a member of an enterprise with managed users, you can only invite other members of your enterprise to collaborate with you. For more information, see "[Types of GitHub accounts."](https://docs.github.com/en/get-started/learning-about-github/types-of-github-accounts#enterprise-managed-users)

**Note:** GitHub limits the number of people who can be invited to a repository within a 24hour period. If you exceed this limit, either wait 24 hours or create an organization to collaborate with more people.

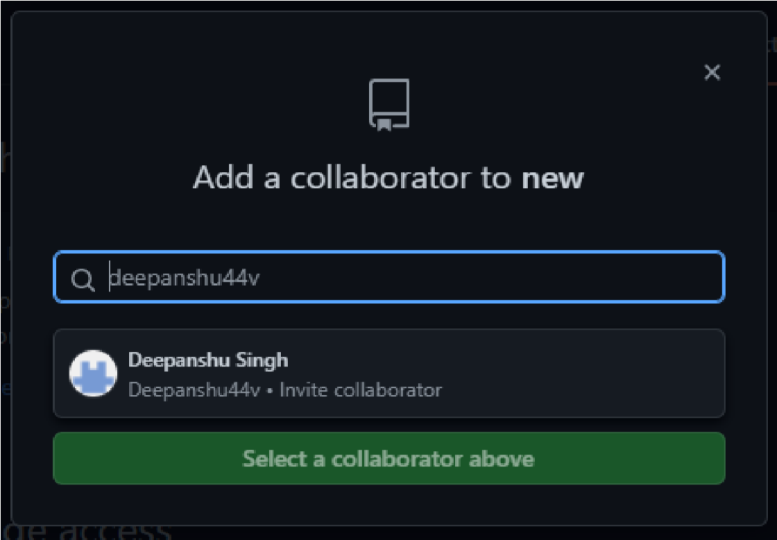
1. Ask for the username of the person you're inviting as a collaborator. If they don't have a username yet, they can sign up for GitHub For more information, see "[Signing up for a new GitHub account".](https://docs.github.com/en/articles/signing-up-for-a-new-github-account)
2. On GitHub.com, navigate to the main page of the repository.
3. Under your repository name, click **Settings**.



1. In the "Access" section of the sidebar, click  **Collaborators & teams**.
2. Click **Invite a collaborator**.



1. In the search field, start typing the name of person you want to invite, then click a name in the list of matches.



1. Click **Add NAME to REPOSITORY**.
2. The user will receive an email inviting them to the repository. Once they accept your invitation, they will have collaborator access to your repository.

**Task:2**

**Fork and Commit**

## • Fork

A fork is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

Most commonly, forks are used to either propose changes to someone else's project to which you do not have write access, or to use someone else's project as a starting point for your own idea. You can fork a repository to create a copy of the repository and make changes without affecting the upstream repository.

Forking a repository is *really* straightforward:

1. Make sure you’re logged into GitHub with your account.
2. Find the GitHub repository with which you’d like to work.



3.



Click the Fork button

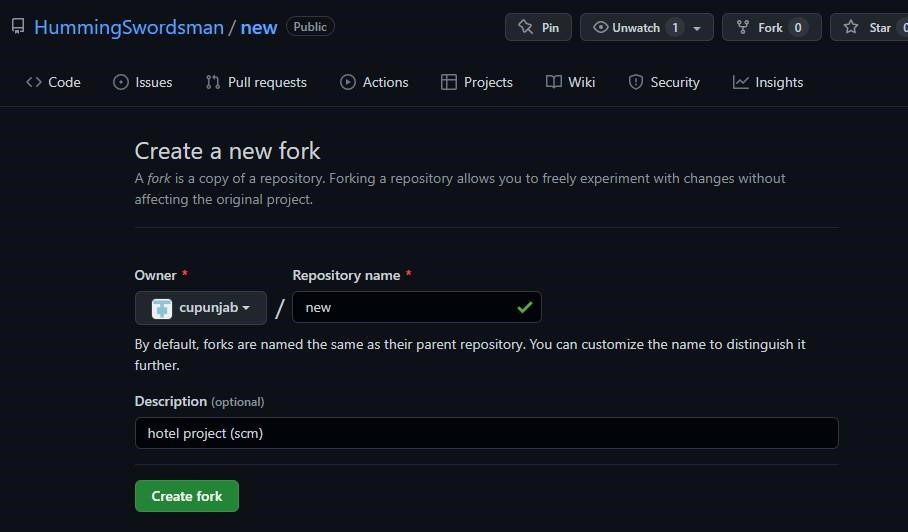
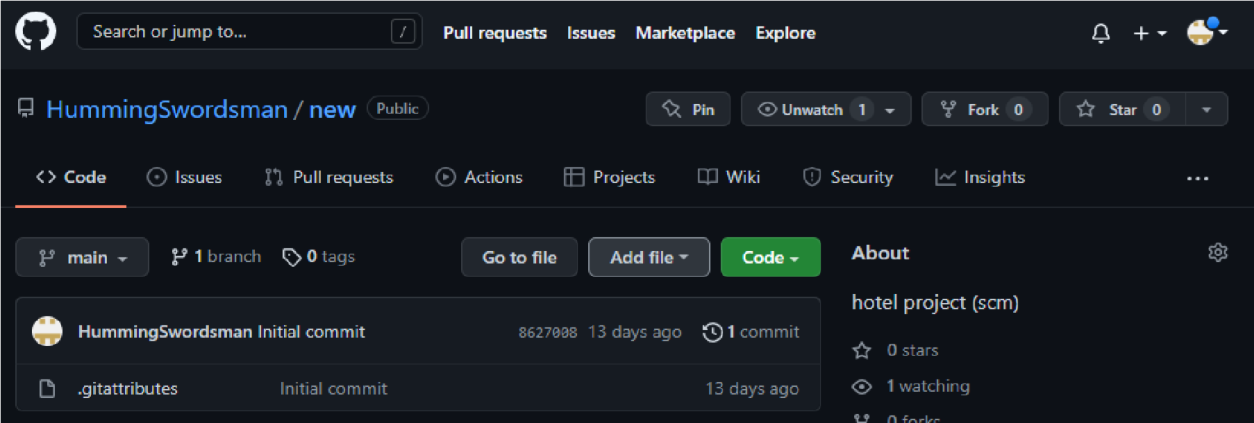
on the upper right



-



hand side of the repository’s page.



That’s it—you now have a copy of the original repository in your GitHub account.

## • Commit

It is used to record the changes in the repository. Every commit contains the index data and the commit message. Every commit forms a parent-child relationship. When we add a file in Git, it will take place in the staging area. A commit command is used to fetch updates from the staging area to the repository.

The git commit command captures a snapshot of the project's currently staged changes. Committed snapshots can be thought of as “safe” versions of a project—Git will never change them unless you explicitly ask it to. Prior to the execution of git commit, The [git add](https://www.atlassian.com/git/tutorials/saving-changes) command is used to promote or 'stage' changes to the project that will be stored in a commit. These two commands git commit and git add are two of the most frequently used.

The commit command without any argument will open the default text editor and ask for the commit message. We can specify our commit message in this text editor. It will run as follows:

**1. $ git commit**

**Task:3**

**Merge and Resolve conflicts created due to own activity and collaborators activity**

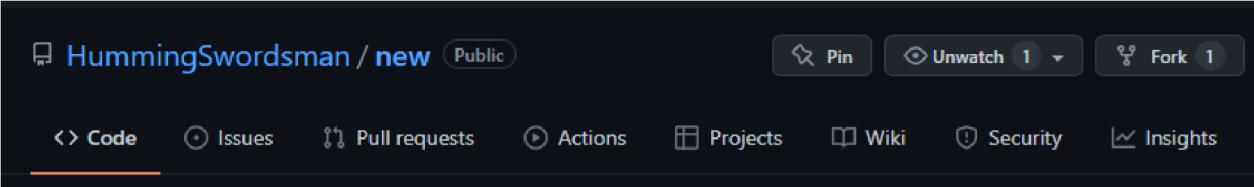
Merging and conflicts are a common part of the Git experience. Conflicts generally arise when two people have changed the same lines in a file, or if one developer deleted a file while another developer was modifying it. In these cases, Git cannot automatically determine what is correct. Conflicts only affect the developer conducting the merge, the rest of the team is unaware of the conflict. Git will mark the file as being conflicted and halt the merging process. It is then the developers' responsibility to resolve the conflict.

How to Resolve Merge Conflicts in Git?

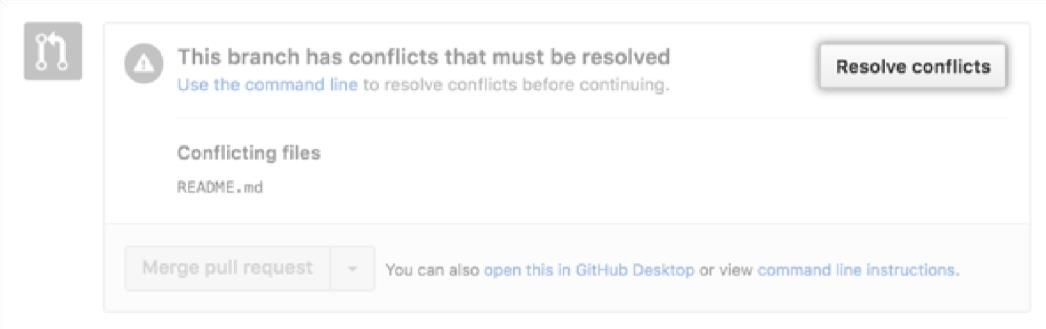
You can resolve simple merge conflicts that involve competing line changes on GitHub, using the conflict editor.

You can only resolve merge conflicts on GitHub that are caused by competing line changes, such as when people make different changes to the same line of the same file on different branches in your Git repository. For all other types of merge conflicts, you must resolve the conflict locally on the command line.

1. Under your repository name, click Pull requests.



1. In the "Pull Requests" list, click the pull request with a merge conflict that you'd like to resolve.
2. Near the bottom of your pull request, click Resolve conflicts.



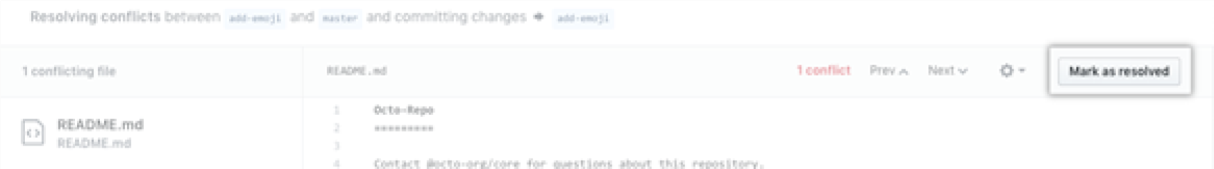
Tip: If the Resolve conflicts button is deactivated, your pull request's merge conflict is too complex to resolve on GitHub. You must resolve the merge conflict using an alternative Git client, or by using Git on the command line. For more information see "Resolving a merge conflict using the command line."

1. Decide if you want to keep only your branch's changes, keep only the other branch's changes, or make a brand new change, which may incorporate changes from both branches.

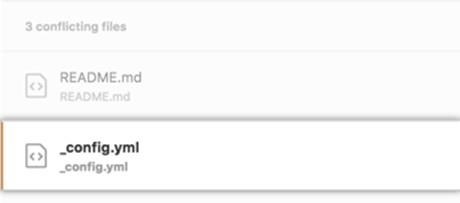
Delete the conflict markers <<<<<<<, =======, >>>>>>> and make the changes you want in the final merge.



1. If you have more than one merge conflict in your file, scroll down to the next set of conflict markers and repeat steps four and five to resolve your merge conflict.
2. Once you've resolved all the conflicts in the file, click Mark as resolved.



1. If you have more than one file with a conflict, select the next file you want to edit on the left side of the page under "conflicting files" and repeat steps four through seven until you've resolved all of your pull request's merge conflicts.



1. Once you've resolved all your merge conflicts, click Commit merge. This merges the entire base branch into your head branch.

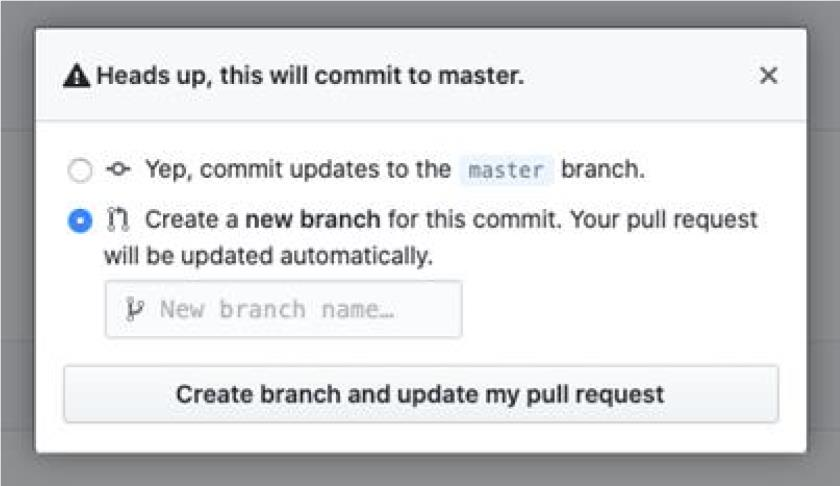


1. If prompted, review the branch that you are committing to. If the head branch is the default branch of the repository, you can choose either to update this branch with the changes you made to resolve the conflict, or to create a new branch and use this as the head branch of the pull request.

If you choose to create a new branch, enter a name for the branch. If the head branch of your pull request is protected you must create a new branch. You won't get the option to update the protected branch.

Click Create branch and update my pull request or I understand, continue updating BRANCH.

The button text corresponds to the action you are performing.



1. To merge your pull request, click Merge pull request. For more information about other pull request merge options, see "Merging a pull request."

**Task:4**

**Reset And Revert**

## **Git Reset**

The git reset command allows you to RESET your current head to a specified state. You can reset the state of specific files as well as an entire branch. This is useful if you haven't pushed your commit up to GitHub or another remote repository yet.

### Reset a file or set of files

The following command lets you selectively choose chunks of content and revert or unstage it.

git reset (--patch | -p) [tree-ish] [--] [paths]

### Unstage a file

If you moved a file into the staging area with git add, but no longer want it to be part of a commit, you can use git reset to unstage that file:

git reset HEAD FILE-TO-UNSTAGE

The changes you made will still be in the file, this command just removes that file from your staging area.

### Reset a branch to a prior commit

The following command resets your current branch's HEAD to the given COMMIT and updates the index. It basically rewinds the state of your branch, then all commits you make going forward write over anything that came after the reset point. If you omit the MODE, it defaults to --mixed:

git reset MODE COMMIT

The options for MODE are:

* --soft: does not reset the index file or working tree, but resets HEAD to commit.

Changes all files to "Changes to be commited"

* --mixed: resets the index but not the working tree and reports what has not been updated
* --hard: resets the index and working tree. Any changes to tracked files in the working tree since commit are discarded
* --merge: resets the index and updates the files in the working tree that are different between commitand HEAD, but keeps those which are different between the index and working tree
* --keep: resets index entries and updates files in the working tree that are different between commitand HEAD. If a file that is different between commit and HEAD has local changes, the reset is aborted

Be very careful when using the --hard option with git reset since it resets your commit, staging area and your working directory. If this option is not used properly then one can end up losing the code that is written.

## **Git Revert**

Both the git revert and git reset commands undo previous commits. But if you've already pushed your commit to a remote repository, it is recommended that you do not use git reset since it rewrites the history of commits. This can make working on a repository with other developers and maintaining a consistent history of commits very difficult.

Instead, it is better to use git revert, which undoes the changes made by a previous commit by creating an entirely new commit, all without altering the history of commits.

### Revert a commit or set of commits

The following command lets you revert changes from a previous commit or commits and create a new commit.

git revert [--[no-]edit] [-n] [-m parent-number] [-s] [-S[<keyid>]] <commit>… git revert --continue

git revert --quit git revert --abort

**Common options:**

-e

--edit

* This is the default option and doesn't need to be explicitly set. It opens your system's default text editor and lets you edit the new commit message before commit the revert.
* This option does the opposite of -e, and git revert will not open the text editor. ●

This option prevents git revert from undoing a previous commit and creating a new one. Rather than creating a new commit, -n will undo the changes from the previous commit and add them to the Staging Index and Working Directory.

--no-edit

-n

-no-commit

**Example.**

Let's imagine the following situation: 1.) You are working on a file and you add and commit your changes. 2.) You then work on a few other things, and make some more commits. 3.) Now you realize, three or four commits ago, you did something that you would like to undo - how can you do this?

You might be thinking, just use git reset, but this will remove all of the commits after the one you would like to change - git revert to the rescue! Let's walk through this example:

mkdir learn\_revert # Create a folder called `learn\_revert`

cd learn\_revert # `cd` into the folder `learn\_revert` git

init # Initialize a git repository

touch first.txt # Create a file called `first.txt` echo Start

>> first.txt # Add the text "Start" to `first.txt`

git add . # Add the `first.txt` file

git commit -m "adding first" # Commit with the message "Adding first.txt"

echo WRONG > wrong.txt # Add the text "WRONG" to `wrong.txt` git

add . # Add the `wrong.txt` file

git commit -m "adding WRONG to wrong.txt" # Commit with the message "Adding WRONG

to wrong.txt"

echo More >> first.txt # Add the text "More" to `first.txt` git

add . # Add the `first.txt` file git commit -m "adding More to first.txt" # Commit with the message "Adding More to first.txt"

echo Even More >> first.txt # Add the text "Even More" to `first.txt` git

add . # Add the `first.txt` file git commit -m "adding Even More to First.txt" # Commit with the message "Adding More to first.txt"

# OH NO! We want to undo the commit with the text "WRONG" - let's revert! Since this commit was 2 from where we are not we can use git revert HEAD~2 (or we can use git log and find the SHA of that commit)

git revert HEAD~2 # this will put us in a text editor where we can modify the commit message.

ls # wrong.txt is not there any more!

git log --oneline # note that the commit history hasn't been altered, we've just added a new commit reflecting the removal of the `wrong.txt`

And with that you're one step closer to getting your black belt in Git.