Git and GitHub workflow challenge

Introduction

The purpose of this challenge is to teach some advanced Git skills that are frequently used when a team of developers are working together on a project.

This challenge must be completed by the end of the 1st week of the Beyond course. The submission date is Friday, March 10th, 2023, at 12:00. The submission form is here.

Git is a complex and versatile tool, and there is frequently more than one way to perform a particular action. In this document, we tried to focus on techniques that allow you to apply the same Git command in a wide variety of situations, but you may find other ways if you search the internet.

General Instructions

- For the purpose of performing this challenge, you will have to pair-up with another student.
- There are two roles to be performed by the students - The role of a project maintainer and the role of a contributor.
- Each of the students must perform both roles, this means that each pair of students is performing this challenge in its entirety twice, playing different roles each time.
- The maintainers will create a “main” or “upstream” repository in their personal GitHub accounts.
- The contributors will “fork” these repositories into their personal GitHub account.
- Only the contributor should submit the results.

Prerequisites

- The Beyond Tool introduction challenge must be completed by the students before performing this one. This challenge assumes that a student is already familiar with the Git techniques shown there.
- Git installed
- A personal GitHub account

Time management

This challenge includes a relatively large number of sections, but students are given a little less than a week to complete it. Therefore, some time planning is required to do it effectively.

We expect that the most time-consuming aspect of this challenge will be when one student has to wait for the other student to complete certain tasks before they can continue their work. Therefore, we recommend that students plan their time and try to reduce the amount of time that they need to hand-off to the other student as much as possible.

With the way this challenge is built, it's actually possible to perform it in four phases:

1. In the first phase, the “maintainer” creates the repository and populates it with initial commits
2. In the 2nd phase, the “contributor” performs a set of tasks to create their various contributed PRs.
3. In the 3rd phase, the “maintainer” modifies their repository in ways that have an impact on the PRs the “contributor” had created
4. In the 4th (and last) phase, the “contributor” reacts to changes made by the “maintainer”.

While this challenge has four parts, they do not align with the phases described above, if you do work according to the plan above, you will effectively perform the various parts “in parallel”.

In summary, read the challenge in its entirety before you begin, and plan your time, so you can use it effectively.

Part 1: Basic repository setup

Maintainer task 1: Create an empty GitHub repository

The maintainer should create a new repository in their GitHub account. The repository name should be as follows:

`beyond-git-challenge-`MNAME

Where “MNAME” would be the maintainer’s GitHub username. The repository should be marked as “Public” and be initialized with a README file, but no “.gitignore” or license file.

If you don’t remember how to create a git repository, please refer to Part 1/Task 2 of the Beyond tool introduction challenge.

Maintainer task 2: Create an initial set of files

To create an initial set of files, the maintainer must first clone the repository to their local computer with the following command

```
git clone https://github.com/MNAME/beyond-git-challenge-MNAME.git
```

(Make sure you replace “MNAME” in the command above with the maintainer’s account name)

Don’t forget to “cd” to a suitable directory before cloning so you’ll know where to edit your files.

After cloning the repository, create the following files in the directory in which it was cloned into with the specified content:

File name	Content

file1.txt	`Suspendisse malesuada eros rutrum tempus efficitur. Sed sollicitudin varius leo sit amet tempus. Quisque consequat justo scelerisque leo ultrices`
	
	`at scelerisque ipsum ultricies. Pellentesque libero dui, luctus vitae quam euismod, lacinia ornare turpis. Donec at viverra massa. Vestibulum laoreet dolor`
	
	`id felis hendrerit, vel mattis dolor vulputate. Nullam ut sagittis nunc. Vivamus lacinia augue sit amet turpis commodo, id dapibus odio dapibus.`

file2.txt	`Aliquam elementum, turpis eu lacinia faucibus, diam elit sollicitudin nunc, sit amet sodales leo sapien et sem.`

	Maecenas mauris eros, sollicitudin at elit in, pellentesque mattis orci. Aliquam elementum finibus dictum. Aenean vitae condimentum eros. Nullam quis laoreet metus. Phasellus at tincidunt dolor. Orci varius natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Cras pretium ullamcorper quam sit amet dapibus. Integer ac nibh nec est porttitor semper ultrices eu ligula. Etiam sodales porttitor venenatis.
file3.txt	Quisque nec lectus viverra, tempus mi molestie, posuere mauris. Nam iaculis ac eros eget bibendum. Phasellus egestas fringilla libero sed viverra. Proin hendrerit convallis nibh ac gravida. Integer placerat accumsan nibh, vitae bibendum urna ullamcorper in.

Please make sure you create the files exactly as shown here. Make sure you break the lines at the same locations and that you don't introduce extra whitespace characters. It is recommended to copy and paste the content.

After creating the files, add and commit them to Git with the following commands (Don't forget to `cd` into the directory before trying these commands):

```
git add file1.txt file2.txt file3.txt
git commit -m "initial set of files"
```

We will now push the commit we just created directly into the "main" branch of our Git repository. This is something that a maintainer can typically do, but that is usually avoided in real software projects because it bypasses the code review and CI mechanisms.

```
git push origin
```

Once we're done pushing, we should be able to see the new files from the GitHub UI.

Contributor task 1: Fork the repository

Once the repository and the initial set of files are created by the maintainer, the contributor can fork the repository to their own GitHub account.

To do this the contributor must navigate to the maintainer's repository on GitHub and click the “Fork” button that is found at the top-right corner. The maintainer’s repository should be found in a URL like the following:

`https://github.com/MNAME/beyond-git-challenge-MNAME`

(Where “MNAME” is the maintainer’s GitHub account name)

After performing the “Fork” the contributor would have their own copy of the repository in their account. That URL for that copy would be:

`https://github.com/CNAME/beyond-git-challenge-MNAME`

(Where “MNAME” is the maintainer’s GitHub account name and “CNAME” is the contributor’s account name)

Contributor task 2: Clone the repository locally

In order to make modifications to the repository files, the contributor needs to clone their copy of the repository to the local machine. This is done with the following command:

```
git clone https://github.com/CNAME/beyond-git-challenge-MNAME.git
```

Please note that we are first cloning the forked copy and not the maintainer’s original repository.

Don’t forget to “cd” to a suitable directory before cloning, so you’ll know where to edit your files.

Contributor task 3: Setup remotes

In order to contribute to the maintainer’s project, the contributor needs a way to fetch updated code from the maintainer’s repository when it becomes available. This is done by configuring an additional *remote* in the local repository.

A “remote” is the object that Git uses to manage communications with repositories that reside on other computers, such as the GitHub servers. Remote definitions contain names that we can use when performing communications against the remote repositories and URLs against which communications are performed. When we clone a repository, Git sets up the “origin” remote for us automatically to point to the remote repository we cloned from. In the “fetch”, “pull” and “push” commands you have seen previously, we have used “origin” to refer to that default setting.

We are going to add another remote configuration to our local repository that is called “upstream”. We will make it point to the maintainer’s repository. This is done with the following command:

```
git remote add upstream https://github.com/MNAME/beyond-git-challenge-MNAME.git
```

(Make sure you replace “MNAME” in the command above with the maintainer’s account name) After running this command we can run the following to view our currently configured remotes:

```
git remote -v
```

The output we will get would look like the following:

```
origin    git@github.com:CNAME/beyond-git-challenge-MNAME.git (fetch)
origin    git@github.com:CNAME/beyond-git-challenge-MNAME.git (push)
upstream  git@github.com:MNAME/beyond-git-challenge-MNAME.git (fetch)
upstream  git@github.com:MNAME/beyond-git-challenge-MNAME.git (push)
```

We can see both our new “upstream” remote as well as the “origin” remote that was created for us. Each remote actually appears twice because it is possible to set up different URLs for reading and writing, that kind of setup is sometimes needed in certain server configurations.

We will now run a `fetch` command to communicate with the remote we’ve just added and fetch information from it:

```
git fetch upstream
```

If everything goes well, the output of the command should look like the following:

```
From https://github.com/MNAME/beyond-git-challenge-MNAME
 * [new branch]    main         -> upstream/main
```

This tells us that Git has successfully communicated with the “upstream” remote and had fetched information about the branches found there (Currently just the “main” branch)

When we run “fetch” Git creates or updates local copies of the branches in the remote repository. The branch names are prefixed with the remote name. That way, the copy of the “main” branch in the “upstream” remote is called “upstream/main”. We can use that to refer to it and we will see it shows up in various places in the Git output.

Part 2: PR history editing

Contributor task 1: Create a new PR

To learn about how to deal with various aspects of the life cycle of a pull request (PR) the contributor will create a new PR that has 3 different commits in it. To do that, 3 new files need to be created in the repository clone directory on the contributor’s computer. File names and contents are described in the following table:

File name	Content
contribution1.txt	`Mauris tortor ligula, vestibulum eget metus eget, vulputate rutrum lectus. Suspendisse ac hendrerit nunc. Nulla varius lectus vitae massa auctor feugiat. Nulla mattis lacus leo, in egestas ex eleifend vitae.` `Praesent ullamcorper eros a mauris lacinia, eu ullamcorper tortor dapibus. Sed eget vehicula felis. Donec vitae ipsum id risus condimentum consequat in a mi. Integer sollicitudin sit amet nunc vitae sollicitudin. Duis tincidunt sapien id ultrices pretium. Nulla mauris sapien, vestibulum nec tortor a, consectetur pharetra libero.`
contribution2.txt	`Nam suscipit ante nec eros consectetur pretium. Etiam lacinia blandit molestie. Nam pulvinar nibh elit, vel condimentum ante accumsan a. Vivamus tellus lectus, pretium nec erat id, porttitor consequat sapien.` `Pellentesque porttitor tincidunt metus. Donec vitae fringilla nulla. Praesent pretium mollis erat, et vestibulum nisi tempus a.`
contribution3.txt	`Donec diam ex, interdum vel leo sed, venenatis molestie ex. Proin fringilla justo vitae ex suscipit rutrum. Proin nec elit tortor.` `Mauris rutrum, felis nec consectetur interdum, nisi mi hendrerit lectus, aliquam accumsan nisl eros sed magna. Maecenas convallis faucibus arcu, vel commodo elit molestie non. In dapibus varius pulvinar.`

After creating the files, the following commands should be run to create a separate commit for each file:

```
git add contribution1.txt
git commit -m "Adding contribution1.txt"
git add contribution2.txt
git commit -m "Adding contribution2.txt"
git add contribution3.txt
git commit -m "Adding contribution3.txt"
```

If we run `git log --oneline --decorate` following the chain of commands above, we should see output like the following:

```
c88f6f3 (HEAD, main) Adding contribution3.txt
651bb5a Adding contribution2.txt
086d304 Adding contribution1.txt
1a2f8b7 (upstream/main, origin/main, origin/HEAD) Initial set of files
fb4cbdd Initial commit
```

Keen eyed observers might notice we've made a common mistake - we've created the new set of commits in the local "main" branch instead of creating a feature branch for them. Let's fix that!

First, we will create a new feature branch as a copy of the current "main" branch. We're using the "branch" rather than the "checkout" command to create the branch, but remain on the "Main" branch for now.

```
git branch contributions
```

We will run a `git log --oneline --decorate` command to view the current status:

```
c88f6f3 (HEAD -> main, contributions) Adding contribution3.txt
651bb5a Adding contribution2.txt
086d304 Adding contribution1.txt
1a2f8b7 (upstream/main, origin/main, origin/HEAD) Initial set of files
fb4cbdd Initial commit
```

We can see a new "contributions" branch was created while "HEAD" (The marker of "where" we are) still points to the "main" branch.

We will now reset our local “main” branch to be identical to the “main” branch of the “upstream” remote repository (true to the last time we’ve run “fetch”).

```
git reset --hard upstream/main
```

The output of the command will indicate we’ve moved to another commit:

```
HEAD is now at 1a2f8b7 Initial set of files
```

(The shortened commit SHA that will be shown in your output will most probably be different than the one shown here)

If we now run the `git log --oneline --decorate` command, the output will resemble the following:

```
1a2f8b7 (HEAD -> main, upstream/main, origin/main, origin/HEAD) Initial set of files
fb4cbdd Initial commit
```

We no longer see the new set of commits because they are no longer in the “main” branch where we currently reside, but no worries! The commits are still there in the “contributions” branch. Let's switch over to it to check:

```
git checkout contributions
```

The output of `git log --oneline --decorate` should now resemble the following:

```
c88f6f3 (HEAD -> contributions) Adding contribution3.txt
651bb5a Adding contribution2.txt
086d304 Adding contribution1.txt
1a2f8b7 (upstream/main, origin/main, origin/HEAD, main) Initial set of files
fb4cbdd Initial commit
```

We can now see that all the branches point to the right commits.

Let’s now continue with the process of creating a PR with our new commits, the new “contributions” branch needs to be pushed to the contributor’s forked copy in GitHub:

```
git push origin -u contributions
```

There are three things to note about the command above:

- We mention “origin”, the name of the remote we’re pushing.
- The “-u” parameter makes Git remember the relationship between the local “contributions” branch and the “contributions” branch this command creates on the remote repository; this will allow us to use a shorter command to update the remote branch in the future.

- We can specify the names of both the local and the remote branches, but here we specify just one of them, so Git just assumes the names are identical. Some versions of Git may print a warning about making that assumption.

The output of the command will resemble the following:

```
Counting objects: 10, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 1.49 KiB | 0 bytes/s, done.
Total 9 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), done.
remote:
remote: Create a pull request for 'contributions' on GitHub by visiting:
remote:
https://github.com/CNAME/beyond-git-challenge-MNAME/pull/new/contributions
remote:
To git@github.com:CNAME/beyond-git-challenge-ifb.git
 * [new branch]    contributions -> contributions
Branch contributions set up to track remote branch contributions from origin.
```

We can see GitHub is being helpful and giving us a link to a page that will create a PR out of the new branch, we will do that in a second but let's first inspect our local repository with the `git log --oneline --decorate` command:

```
c88f6f3 (HEAD -> contributions, origin/contributions) Adding contribution3.txt
651bb5a Adding contribution2.txt
086d304 Adding contribution1.txt
1a2f8b7 (upstream/main, origin/main, origin/HEAD, main) Initial set of files
fb4cbdd Initial commit
```

We can see we now have a local copy of the “contributions” that we’ve created in the “origin” remote repository. The copy is called “origin/contributions”. It is currently identical to our local “contributions” branch, but the two will diverge as we make further changes.

Let’s now click on the link GitHub gave us to create the new PR. This will lead us to the “Open pull request” screen. We will typically take the time to write down a detailed description for our pull request, but for now we can just accept the default title GitHub filled-in for us and click on “Create pull request”.

In the pull request screen we can now click on the “commits” tab and see the three commits that were included as part of this pull request, we can also click on each commit and see the file that was included in it.

Contributor task 2: Fixing a file in the PR

Let's say that the project maintainers and other contributors have reviewed the PR and have suggested that we make some changes to the “contribution1.txt” file. Let's go ahead and do those.

First, let's make sure we're working on the right branch. We do this by running the `git status` command. The output looks like the following:

```
# On branch contributions
nothing to commit, working directory clean
```

We can see that we’re on the “contributions” branch which is the one we created the PR from, if that is not the case we can check it out with `git checkout contributions`.

Now we can make changes to the “contribution1.txt” file. Use a text editor to make changes for its content, for example, remove a few lines or add them. Once we are done with that, let's commit the change:

```
git add contribution1.txt
git commit -m "Fixing contribution1.txt"
```

If we look at `git log --oneline --decorate` we can see the new commit we just added:

```
32eaa7c (HEAD -> contributions) Fixing contribution1.txt
c88f6f3 (origin/contributions) Adding contribution3.txt
651bb5a Adding contribution2.txt
086d304 Adding contribution1.txt
1a2f8b7 (upstream/main, origin/main, origin/HEAD, main) Initial set of files
fb4cbdd Initial commit
```

We can also see that, unsurprisingly, our local “contributions” branch had diverged from the copy we’ve pushed to the contributor’s repo in GitHub. Let's fix that by pushing the branch again:

```
git push
```

Since we’ve used the `-u` parameter when we ran `git push` for this branch previously, we can avoid specifying any parameters to the command.

Since we’ve made a PR to the maintainer’s repo that references the branch we’ve pushed to, updating the branch causes the PR to be updated as well. We can see that on the PR screen. The “commits" tab should now show 4 commits, and the “changed files” tab should show the new state of the “contribution1.txt” file.

Contributor task 3: Squashing a fix commit

The PR sent by the contributor is a little "dirty" right now. It introduces the "contribution1.txt" file with a certain content, and then modifies its content shortly after. We would like the PR to introduce the file with its final content.

To do this, we need to combine our first and last commits together. In Git terminology, this is called "squashing" those commits together.

To squash the commits, we must first run `git status` to make sure we're on the "contributions" branch, and then we run the following command:

```
git rebase -i main
```

The command will open a new text editor to let us provide more input into the process, but before we do that, let's have a closer look at the command and its parameters. The `rebase` command is a versatile command that can do many different things. The `-i` option lets us perform various *interactive* edits to commits in the branch we're working on. We also need to specify the starting point for the set of commits we'll edit. Here, we specify "main" to indicate that we want to edit all the commits in our current branch ("contributions") since the point in which it diverged from the "main" branch.

The contents of the text editor Git opens will look like the following:

```
pick 086d304 Adding contribution1.txt
pick 651bb5a Adding contribution2.txt
pick c88f6f3 Adding contribution3.txt
pick 32eaa7c Fixing contribution1.txt
```

(Git will also include some comments showing helpful instructions about what we need to do, take some time to look at them).

What we see in the editor is the list of commits we've chosen to edit. What we'll need to do is move the commit that fixes the "contribution1.txt" file to directly follow the one that added it and then mark it to be squashed with it. When we finish our editing the file should look like so:

```
pick 086d304 Adding contribution1.txt
squash 32eaa7c Fixing contribution1.txt
pick 651bb5a Adding contribution2.txt
pick c88f6f3 Adding contribution3.txt
```

We can now save the file and exit the editor to make Git continue with the operation.

What Git will do is immediately open another editor window. This time, it's meant to allow us to edit the commit message of the final result. The contents of the editor will resemble the following:

```
# The first commit's message is:

Adding contribution1.txt

# This is the 2nd commit message:

Fixing contribution1.txt
```

The editor includes the commit messages from all the commits we're squashing together. In this case, we only need the message from the first commit, so we'll edit the contents to look like the following:

```
Adding contribution1.txt
```

Now, when we save the file and exit the editor, the process will be completed. We can inspect the results with `git log --oneline --decorate` command:

```
0634366 (HEAD -> contributions) Adding contribution3.txt
cde9e23 Adding contribution2.txt
9dfbf9e Adding contribution1.txt
1a2f8b7 (upstream/main, origin/main, origin/HEAD, main) Initial set of files
fb4cbdd Initial commit
```

We can see now that the local "contributions" branch only contains three commits. This seems pretty similar to how things were when we created the branch in Task 1 above, but if we look carefully, we'll notice a few important differences:

- The SHA numbers are different from what they were, this is because Git actually generated a new set of commits to accommodate the changes we made.
- We don't see any mention of the remote "origin/contributions" branch in the log output. This is because our local "contributions" branch had now fully diverged from it.

Let's go ahead and update the remote branch and the PR:

```
git push
```

Running this command will not work as expected, we will get an error message like the following from Git:

```
To git@github.com:CNAME/beyond-git-challenge-MNAME.git
 ! [rejected]      contributions -> contributions (non-fast-forward)
error: failed to push some refs to
 'git@github.com:CNAME/beyond-git-challenge-MNAME.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

The reason for this issue is that our local “contributions” branch we’re pushing from is no longer derived from the “contributions” branch on the “origin” repo that we’re pushing into. Which is to say, the topmost commit in the “contributions" branch on the “origin” repo is no longer included in the local “contributions" branch.

We will not go into details about why Git prevents this kind of push by default, you can read some of those in the help page that is mentioned in the error message. For now it's sufficient to say that this is a safety mechanism for situations where multiple contributors edit the same branch directly. This is not usually the case when making modifications via the GitHub PR mechanism, and therefore its usually safe to bypass this protection with the following command:

```
git push -f
```

This time, the push process should be completed successfully. If you look carefully at the output, you will notice Git is mentioning it is performing a “forced update”, that is the meaning of the `-f` flag.

If we look at `git log --oneline --decorate` now:

```
0634366 (HEAD -> contributions, origin/contributions) Adding contribution3.txt
cde9e23 Adding contribution2.txt
9dfbf9e Adding contribution1.txt
1a2f8b7 (upstream/main, origin/main, origin/HEAD, main) Initial set of files
fb4cbdd Initial commit
```

We can see the “origin/contributions” remote branch has now fully converged with the local branch.

Go ahead and inspect the PR. Check that the commits you see in the PR reflect the changes we have made.

Contributor task 4: Putting commits together

Let's assume the contributor takes another look at the set of commits in the “contributions” branch and decides that the “contribution2.txt” and “contribution3.txt” should reside in the same commit.

Making this change is quite similar to what we did before. After making sure that we’re on the “contributions” branch, we use the `git rebase -i main` command and get to the editor window with the following content:

```
pick 9dfbf9e Adding contribution1.txt
pick cde9e23 Adding contribution2.txt
pick 0634366 Adding contribution3.txt
```

We will edit the contents to indicate that the last commit should be squashed with the second:

```
pick 9dfbf9e Adding contribution1.txt
pick cde9e23 Adding contribution2.txt
fixup 0634366 Adding contribution3.txt
```

Note that we’re using “fixup” rather than “squash” here. Their functionality is pretty similar, the only difference is that using “fixup” skips the commit message editing phase and just uses the message of the first commit in the set of commits that are squashed together.

After saving and exiting the text editor, we can inspect the results with `git log --oneline --decorate`:

```
dd861c2 (HEAD -> contributions) Adding contribution2.txt
9dfbf9e Adding contribution1.txt
1a2f8b7 (upstream/main, origin/main, origin/HEAD, main) Initial set of files
fb4cbdd Initial commit
```

We can see the local “contributions” branch now only contains two commits. We can also see that it is once more fully diverged from the remote branch. This is expected to happen every time we use the “git rebase” command to edit commits.

Let’s go ahead and push local branch content to the remote one:

```
git push -f
```

Contributor task 5: Splitting a commit apart

The maintainers have looked at the PR after the last round of changes we’ve made. They have decided that putting the “contribution2.txt” and “contribution3.txt” files in the same commit was not such a good idea after all. We are going to have to split them apart.

Splitting a commit apart involves a process where we first tell Git to “forget” the commit while still keeping all the actual changes and files we need locally, and then re-adding and re-committing the files as needed to create the desired new chain of commits.

We begin by telling Git to “forget” the topmost commit in the “contributors” branch:

```
git reset HEAD~1
```

In the command above, the number indicates the amount of commits we want to go back.

If we review the branch log with `git log --oneline --decorate`:

```
9dfbf9e (HEAD -> contributions) Adding contribution1.txt
1a2f8b7 (upstream/main, origin/main, origin/HEAD, main) Initial set of files
fb4cbdd Initial commit
```

We can see we now only have one commit in the branch. If we however inspect the status of the repository with `git status`:

```
# On branch contributions
# Your branch is behind 'origin/contributions' by 1 commit, and can be
# fast-forwarded.
#   (use "git pull" to update your local branch)
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#    contribution2.txt
#    contribution3.txt
nothing added to commit but untracked files present (use "git add" to track)
```

We can see the “contribuition2.txt” and “contribution3.txt” files are still there but are unknown to Git.

We can now use a sequence of `git add` and `git commit` commands to re-add the files and create new commits as desired:

```
git add contribution2.txt
git commit -m "Adding contribution2.txt"
git add contribution3.txt
git commit -m "Adding contribution3.txt"
```

Inspecting the branch log, we can see we once more have three commits in the branch:

```
1745477 (HEAD -> contributions) Adding contribution3.txt
6979cc8 Adding contribution2.txt
9dfbf9e Adding contribution1.txt
1a2f8b7 (upstream/main, origin/main, origin/HEAD, main) Initial set of files
fb4cbdd Initial commit
```

If we look carefully, we can see that:

- The commit SHAs are different from the ones we had before. This is because we made new commits
- We are once more diverged from the branch in the contributor's repository
- The commit SHA of the first commit actually did not change. This is because we did not make any changes to it.

We can now push our new branch content to GitHub:

```
git push -f
```

Contributor task 6: Moving a commit to a different PR

The project maintainers have reviewed the PR again, and have decided that the addition of the "contribution3.txt" file is not directly related to the "contribution2.txt" and "contribution1.txt" files and therefore should be moved to a different PR.

To move a commit to a new PR we begin by creating a new branch for the new PR:

```
git checkout -b contribution3 main
```

We use the command above to simultaneously perform 3 operations:

- Create the "contribution3" branch
- Make the new branch be identical to the "main" branch (We usually want to make new PR branches be derived from "main" so they could be merged back into it).
- Checkout the new branch so we can begin working on it immediately

We can confirm the results of the command by running `git log --oneline --decorate`:

```
1a2f8b7 (HEAD -> contribution3, upstream/main, origin/main, origin/HEAD, main)
Initial set of files
fb4cbdd Initial commit
```

Now we need to copy the commit containing the “contribution3.txt” file to the new branch. We can use the cherry-pick command to do that:

```
git cherry-pick contributions
```

Since the commit we’re copying is the topmost commit in the “contributions” branch, we can use the branch name to refer to it. If we wanted the 2nd topmost commit we could use `contributions~1` to refer to it, if we wanted the 3rd topmost we could use `contributions~2` and so on. Alternatively, the commit SHA identifier can also be used to identify the desired commit.

We can look at the branch log to see the new commit we’ve just added:

```
7e0876f (HEAD -> contribution3) Adding contribution3.txt
1a2f8b7 (upstream/main, origin/main, origin/HEAD, main) Initial set of files
fb4cbdd Initial commit
```

Note: The cherry-pick command we’ve just ran finished smoothly because the commit we’ve copied is adding a new file and therefore is quite independent from other commits. In real-life scenarios various issues may arise when performing cherry-pick in the form of Git conflicts. Those conflicts can be resolved using the techniques described in Part 3 of this challenge.

Let's push the new branch to the contributor’s fork on GitHub:

```
git push -u origin contribution3
```

We can now use the link GitHub provides us to create a PR from the new branch. After doing so, please take the time to inspect the new PR and check that it only contains a single commit with the “contribution3.txt” file.

Now it's time to remove the commit and the file from the other PR. For this, we must first checkout the relevant branch:

```
git checkout contributions
```

Now we use the “rebase” command to remove the undesired commit:

```
git rebase -i main
```

Since we want to remove a commit we simply delete the line representing it from the file shown in the text editor. The end result should resemble the following:

```
pick 9dfbf9e Adding contribution1.txt
pick 6979cc8 Adding contribution2.txt
```

We save the file and exit the editor to finish the operation.

Again, we can use “git log” to confirm the results:

```
6979cc8 (HEAD -> contributions) Adding contribution2.txt
9dfbf9e Adding contribution1.txt
1a2f8b7 (upstream/main, origin/main, origin/HEAD, main) Initial set of files
fb4cbdd Initial commit
```

Now all we’re left to do is to push the branch so the PR gets updated:

```
git push -f
```

Please take the time to inspect the PR and ensure it now only contains two commits and only the “contribution1.txt” and “contribution2.txt” files.

Note: Since we’ve removed the topmost commit in the “contributions” branch, we could have also used the “git reset” command to remove it. Since we don’t always have to remove the topmost one, we’ve shown how to do this using the “git rebase” command that can be used to remove any commit.

Maintainer task 1: Changing the main branch

Since the review process of the new PR is taking a while, other PRs have been submitted and merged in the meantime. The maintainer will emulate this by pushing a new commit into the “main” branch. This can be done anytime after the contributor’s initial PR is submitted.

To do this, the maintainer should create a file called “**file4.txt**” with the following content:

```
Donec luctus viverra urna, non ullamcorper neque facilisis eget.
Etiam id massa enim. Vestibulum eu velit eu sapien faucibus
ultricies. Maecenas sollicitudin sapien elit.

Sed bibendum velit vulputate libero tempor egestas. Sed maximus
augue bibendum, congue tortor eget, tristique ipsum. Etiam ut
fringilla erat.
```

After creating the file, the maintainer should add, commit and push it to their GitHub repository:

```
git add file4.txt
git commit -m "Adding file4"
git push origin
```

Contributor task 7: Rebasing the PR

Having the main project repository change while a contributor is working on a change is quite a common scenario. Therefore, a contributor should frequently check if changes have been made. Especially before beginning to work on making a new PR or making changes to an existing one.

A local Git repository tracks the status of remote repositories it is connected to, but it only updates its “knowledge” about them when we run commands that communicate with those repositories, such as “git push” or “git pull”. Another useful command is “git fetch” which is meant to be used to directly fetch information from a remote repository. Let run it now (On the contributor’s computer) to check the status on the project repository:

```
git fetch upstream
```

Note that we specify “upstream” here as the name of the remote repository we’re fetching from, that is because on the contributor’s computer we have defined “upstream” to refer to the maintainer’s repository in Part 1/Task 3 above.

The output of the command looks like the following:

```
From https://github.com/MNAME/beyond-git-challenge-MNAME
   1a2f8b7..4fd36e6  main       -> upstream/main
```

The output tells us that the local repository’s tracking-copy of the upstream repository’s “main” branch (`upstream/main`) had been updated to reflect the fact that a new commit had been added to the maintainer’s repository.

Let's check how this change is reflected in `git log --oneline --decorate` output (Assuming we still have the “contributions” branch checked out):

```
6979cc8 (HEAD -> contributions, origin/contributions) Adding contribution2.txt
9dfbf9e Adding contribution1.txt
1a2f8b7 (origin/main, origin/HEAD, main) Initial set of files
fb4cbdd Initial commit
```

This does not look very different from the output we’ve seen before, but if we look closely we’ll see that “upstream/main” no longer appears in the log. This is because it no longer points to a commit that resides in the Git history of the “contributions” branch.

We can still see the “main” and “origin/main” branches in the log because those branches had not (yet) been updated to reflect the status of the “main” branch in the maintainer’s repository. Since we frequently use those branches when running “rebase” operations or creating new PRs, let's go ahead and update them.

To do this, we need to first checkout the “main” branch:

```
git checkout main
```

Now we can update the local “main” branch by using the “pull” command:

```
git pull --ff-only upstream main
```

We need to tell the command the name of the remote (“upstream”) and the name of the branch (“main”) to pull from. The `--ff-only` option is a safety measure to prevent Git from trying to automatically merge commits it is fetching from the remote repository with commits we’ve added to “main” locally. We should generally not add commits to the “main” branch locally, but it's a common mistake to make, so it's a good idea to protect against it. If we did add commits, the “pull” command will fail, and we will need to move the commit to a separate branch as described in Task 1 above.

If the command works successfully, the output will look like the following:

```
From https://github.com/MNAME/beyond-git-challenge-MNAME
 * branch          main         -> FETCH_HEAD
Updating 1a2f8b7..4fd36e6
Fast-forward
 file4.txt | 8 ++++++++
 1 file changed, 8 insertions(+)
 create mode 100644 file4.txt
```

The output tells us that the local “main” branch is updated with fetched information, and it's even listing the exact file changes that have been seen.

Now let's also update the copy of the “main” branch on the contributor’s forked repository in GitHub:

```
git push origin main
```

Looking at the log of the “main” branch should now reflect that all its copies are aligned:

```
4fd36e6 (HEAD -> main, upstream/main, origin/main, origin/HEAD) Adding file4
1a2f8b7 Initial set of files
fb4cbdd Initial commit
```

Since the contributor is in the process of working on a contribution PR, it's a good idea to update it with the latest code to make sure it's compatible with it. Often it is not possible to merge the PR without doing so.

To do this we must first checkout the “contributions” branch from which the contributor’s PR as created:

```
git checkout contributions
```

If we have a quick look at the branch log, we can see that the “main” branch no longer appears in it:

```
cde9e23 (HEAD -> contributions) Adding contribution2.txt
9dfbf9e Adding contribution1.txt
1a2f8b7 Initial set of files
fb4cbdd Initial commit
```

We’re going to fix that using the following “rebase” command:

```
git rebase main
```

This command is used to sync between the main branch and your current branch. It does this by first copying everything in main and then adding the commits that you have done in the current branch on top of them. This ensures that the commit history is identical in all branches. The output is quite detailed and lists all the commits that are being recreated:

```
First, rewinding head to replay your work on top of it...
Applying: Adding contribution1.txt
Applying: Adding contribution2.txt
```

Let's inspect the results with `git log --oneline --decorate`:

```
e44b878 (HEAD -> contributions) Adding contribution2.txt
f66dd23 Adding contribution1.txt
4fd36e6 (upstream/main, origin/main, origin/HEAD, main) Adding file4
1a2f8b7 Initial set of files
fb4cbdd Initial commit
```

We can now see the new commit that was added to the “main” branch in the commit history of our local “contributions” branch. This means the new change was successfully incorporated into our branch.

As with every change that we make to the commit history using the “git rebase” command, we need to use the “-f” option to update the remote branch and the PR in GitHub:

```
git push -f
```

Note that the commit made by the maintainer (the "main" branch that was pulled in during the previous step) and the commits made in the local "contributions" branch did not modify the same files. This makes the “git rebase” command run smoothly. In real software projects that may not always be the case, and running “git rebase" may result in merge conflicts that need to be resolved as shown in Part 3 below. When working on real software projects, this is quite likely to happen as developers work on multiple files to implement different features. Care should be taken to write code in a modular fashion and split into various files in order to reduce the probability of this happening

Note: Typically, for a given PR, the maintainer/code reviewer will make multiple requests for changes. For example, commits need to be modified or squashed, etc. It is highly advisable to always begin by performing a rebase on an updated version of “main” before trying to perform any other changes.

Part 3: Conflict resolution

Note: In this section, the tasks do not include specific examples for how to do each step. This is to give you the opportunity for some self-learning, similar to the experience expected during the course. Please take advantage of communication channels if you cannot figure out how to continue on your own.

Contributor task 1: Introduce file changing PR

As mentioned in Part 2/Task 7 above, in real software development, there are scenarios where code changes made by different developers conflict with each other. We will see how Git helps us detect and deal with those situations.

To create a conflict, we must first have the contributor send a PR.

As the contributor, please create a new branch called “change”. In that branch, create a new commit where you modify the files in the repository according to the following table:

File name	New Content

file1.txt	`Suspendisse malesuada eros rutrum tempus efficitur. Sed sollicitudin varius leo sit amet tempus. Quisque consequat justo scelerisque leo ultrices`
	
	`CHANGED BY CONTRIBUTOR`
	
	`id felis hendrerit, vel mattis dolor vulputate. Nullam ut sagittis nunc. Vivamus lacinia augue sit amet turpis commodo, id dapibus odio dapibus.`

file2.txt	`Aliquam elementum, turpis eu lacinia faucibus, diam elit sollicitudin nunc, sit amet sodales leo sapien et sem. Maecenas mauris eros, sollicitudin at elit in, pellentesque`
	`mattis orci. Aliquam elementum finibus dictum.`
	
	`CHANGED BY CONTRIBUTOR`
	
	`Integer ac nibh nec est porttitor semper ultrices eu ligula. Etiam sodales porttitor venenatis.`

file3.txt	`Quisque nec lectus viverra, tempus mi molestie, posuere mauris. Nam iaculis ac eros eget bibendum. Phasellus egestas fringilla libero sed viverra.`
	
	`Proin hendrerit convallis nibh ac gravida. Integer placerat`
	`accumsan nibh, vitae bibendum urna ullamcorper in.`

	ADDED BY CONTRIBUTOR

Note: when you create the new branch, make sure you base it on the “main” branch and not on any other branches that may be in the contributor’s repository.

After making the changes to the file, commit them and create a new PR containing them.

Maintainer task 1: Push conflicting changes

The maintainer will emulate merging changes into the repository. To do this, the maintainer should make changes to the files in the repository according to the following table:

File name	**New Content**

file2.txt	`Aliquam elementum, turpis eu lacinia faucibus, diam elit sollicitudin nunc, sit amet sodales leo sapien et sem. Maecenas mauris eros, sollicitudin at elit in, pellentesque`
	`mattis orci. Aliquam elementum finibus dictum.`
	
	`CHANGED BY MAINTAINER`
	
	`Integer ac nibh nec est porttitor semper ultrices eu ligula. Etiam sodales porttitor venenatis.`

file3.txt	`Quisque nec lectus viverra, tempus mi molestie, posuere mauris. Nam iaculis ac eros eget bibendum. Phasellus egestas fringilla libero sed viverra.`
	
	`CHANGED BY MAINTAINER`

In addition to making the changes described above, the maintainer should also remove the “file1.txt” file by using the git rm command.

Note: The maintainer should be making the changes directly on their “main” branch.

After making all the changes, commit them and push them to the “main” branch of the maintainer’s repository on GitHub.

Contributor task 2: Rebase PR and resolve conflicts

Now that the maintainer has pushed a new branch to their “main” branch, the contributor should perform the same procedure described in Part 2/Task 7 above to update their copies of the “main” branch and rebase the “change” branch. This time, the output of the “git rebase” command should indicate that conflicts had been detected. The conflicts must be resolved before proceeding.

Git does have some built-in conflict resolution algorithms that it tries to run. Those are sometimes successful even when multiple changes have been done to the same file by different parties. In our case, because changes were made to lines that are very close together, the algorithm fails. The “git rebase” output will report to us various details about the algorithms that were run and their failures.

Take a close look at the instructions that are added at the end of the output message, those tell us how to proceed.

Git has essentially stopped in the middle of the rebase process and is expecting us to intervene manually and fix the conflicts before we use the command that is mentioned in the “git merge” output to make it continue. We can also tell Git to abort the whole process and bring things back to the way they were before we started the rebase.

Our next step is to run `git status` to check the current status of the files in our repository.

What we can see there is that “file3.txt” is included in the set of files to be committed - this is because Git has successfully resolved the conflict in it automatically and is ready to carry-on the merge process with that file.

As for “file1.txt” and “file2.txt”, they both require our manual intervention for different reasons. Git tells us what it knows about the reasons for the files failing to merge in its `git status` output.

Use your text editor and the various Git commands such as “git rm” and “git add” to bring the files to the following desired state:

- Files that have been deleted by the maintainer should remain deleted. There is no point in incorporating the contributor’s changes to files that the maintainer had decided to delete.
- For files where conflicting changes had been made to the content - incorporate the content from both the maintainer and the contributor. Be careful not to leave behind the content origin markers that “git rebase” adds to such files.
- When you are done, all files that we keep should show in green and no file should show up in red.

After manually resolving the conflicts, run the command that you’ve seen in the “git rebase” output to continue the rebase process.

Finally, push your rebased branch to update the PR.

Part 4: Dealing with cross-PR dependencies

Contributor task 1: Introduce a PR

As the contributor, perform the following tasks:

1. Create a new branch based on an updated version of “main”.
2. Add some files to the repository and commit them to the new branch
3. Push the new branch to your fork
4. Create a PR from the new branch

Contributor task 2: Introduce a dependant PR

Again, as the contributor, perform the following tasks:

1. Create another new branch, but this time base it on the branch you’ve created in Task 1 above.
2. Add more files to the repository and commit them.
3. Check “git log”, you should see the branch from Task 1 appear in the git history of the new branch.
4. Push the new branch to GitHub and create a PR.

Inspect the commits in the new PR from the GitHub UI. You should see both the commits from the new branch that you intended to include in this PR, as well as the commits from the branch you created in Task 1. This is a known issue with GitHub, where it fails to understand the dependency relationships between different PRs.

For PRs such as the one you just created that depend on code that is included in other PRs, you should make sure to carefully mention the dependency in the PR description so that other project maintainers and contributors know not to try to merge the PR before first merging the ones it depends on. If the project uses some convention to mark PRs that should not be merged, it should be used as well. Many projects use GitHub labels for this.

Maintainer task 1: Merge the initial PR

As the maintainer, use the GitHub UI to merge the PR that was created in Task 1.

Contributor task 3: Review status in GitHub after merge

Once the PR from Task 1 is merged, take a look at the PR screen for the PR you’ve made in Task 2 above. Even though the commit from the first PR was merged and is now included in the “main” branch, it still appears in the PR as if it wasn't merged.

Use the commands you’ve learned to rebase the PR so that it only contains the new commit.