

Mandatory Assignment 1

Welcome to the World of Git (5 Points)

University of Oslo - IN3110/IN4110

Fall 2020

1.1: Your First Git Repository (1 points)

Task: If you haven't already done it, install git (<https://git-scm.com/>). We already created a private assignment repository for you. Clone your repository. Now, create an `assignment1` folder in your git repository. Add a textfile to the `assignment1` directory. The textfile should be named `myfirstcommit.txt` and contain the string "This is my first commit.". Push your first commit to github.

1.2: Recovering Old Versions of Files (2 points)

With this task you will learn how to recover old versions of a file. Why do we need to know this? Well, sometimes when introducing a new feature you might realize that the original framework actually worked better for your purposes, or you made a mistake and need to debug.

Task: Add, commit and push a new file called `friendly_greeting.txt` containing a friendly greeting to your repository. Then change the file so the greeting is less friendly, and add, commit and push the modified file. Since the greeting is no longer friendly change the name of the text file `friendly_greeting.txt` to `less_friendly_greeting.txt`. Use the command

```
git mv friendly_greeting.txt less_friendly_greeting.txt
```

To fetch the old version, first use

```
git log
```

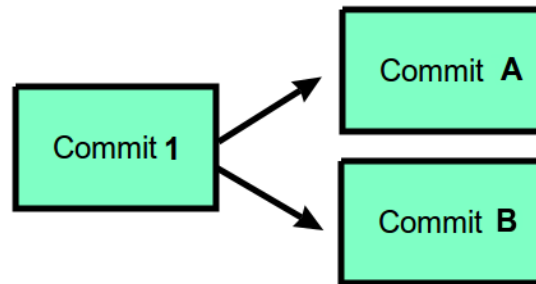
to get a list of the commits, and identify the commit containing the old version of the greeting, and note its commit hash (the string of letters/numbers after "commit " - basically the "name" of the commit). Then use

```
git checkout COMMITHASH friendly_greeting.txt
```

where `COMMITHASH` is the commit hash you found using `git log`. This will recover the friendly version `friendly_greeting.txt` from the commit you chose, and `git add` it for you. Finally, use `git commit` to make a commit which recovers the friendly version, and push this commit to github. Now you will have a friendly and a not friendly greeting version in your repository

1.3: Dealing with Conflicts in Git (2 points)

During your career you will quite likely work with humans. Humans are different, which is fantastic in that sense that everyone brings something to the table. But it also means, that we might encounter conflicts. This exercise will use a lot of words to explain a common “error” which frequently happens when using git to work together. So it is actually a good idea to read through the whole exercise because that type of error will happen and git will complain, even if nobody does anything wrong. But on the sunny side understanding the reasons why git complains makes fixing its complaints a lot less frustrating¹.



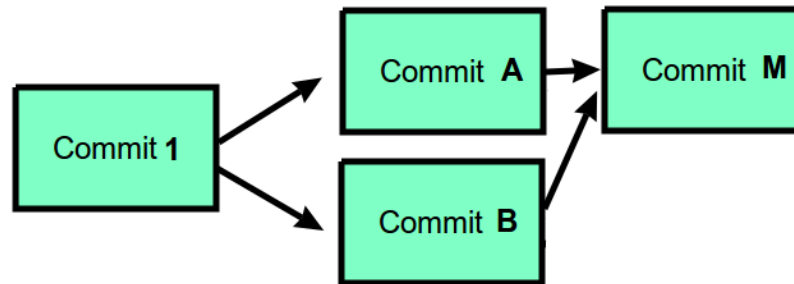
In git, each commit goes sequentially “on top of” another commit. This means that changes made in the commit are *intentional*. However, if two people are working on a repository separately, and both make commits A and B locally and want to push them, git can get concerned². Git will be perfectly happy to accept whichever of A and B is pushed first (let’s say A is first, B is second), but when the other one is pushed, it will not be quite sure what to do with it, and throw an error which can look intimidating.

This is *by design*. While it would sometimes have been possible to just pretend commit B was really supposed to go on top of A - maybe A and B are modifying separate files - git won’t do that by itself, because it is possible that this will have unintended side-effects. Instead, git will inform you that commit A was made “in between” commit 1 and the commit you are trying to make

¹This is also good life advice.

²In a perfect world, each person having their own *branch* would be tidier and minimize the time spent resolving conflicts, but who’s perfect?!

(commit B), and so won't allow you to push³ until you have done a `git pull` to get commit A.



When you do `git pull`, git will see that you are trying to pull commit A which goes on top of commit 1. However, your local copy of the repository already has a commit B on top of commit 1, so completing the `git pull` means you have to `merge` first. This means that you have to make a new `git commit M` which does what commit B did, but on top of commit A instead of commit 1.

Sometimes this is very easy - maybe there is no conflict between B and A at all, and git will intelligently, as it was designed, do it for you. Sometimes, maybe A introduced some new stuff you need to adapt B to work with - maybe a function you use in B was renamed, so you need to change B a bit. In any case, once you have done what needs to be done and made commit M locally, it will basically fit on top of both B and A, and it can be pushed just fine.

Let's try to simulate such a scenario and fix the resulting merge conflict:

1. Add, commit, and push a new, file `gitconflict.txt` with the contents

```
Here's a line

Hello world!

Here's another line
```

to your git repository.
2. Copy the directory on your local machine containing your git repository to a different directory.
3. In the two different local copies of your repository, make different changes (For example, change the line "Hello world!" to "Hello world from A!" in one, and "Hello world from B!" in the other.) to `gitconflict.txt` and commit them separately.

³It's possible to tell git to do the push anyway. In case you didn't guess it yet: This is generally a bad idea.

4. Then, attempt to push the changes of both repositories to github. The first push will be fine, but the second one will fail and tell you the remote has stuff you don't. It will suggest you pull, so do that. This will cause a merge conflict which you will need to resolve manually. Your error will look something like this:

```
Auto-merging gitconflict.txt
CONFLICT (content): Merge conflict in gitconflict.txt
Automatic merge failed; fix conflicts and then commit the result.
```

If you type `git status`, you will see that git started merging stuff automatically, but didn't quite know what to do with `gitconflict.txt`, so git left it in a half-merged state. Opening the copy of `gitconflict.txt` in the second repository shows you something like this:

Here's a line

```
<<<<<< HEAD
Hello world from B!
=====
Hello world from A!
>>>>>> daa83f4b022b0b5b61a40fef6bb8eedacfe9fd5a
```

Here's another line

Notice that the line on which conflicting changes was made has been replaced by some autogenerated text from git. The stuff above `=====` is what commit B wants this line to contain, while the stuff below is what commit A wants this line to contain⁴.

To finish the merge, simply modify the whole chunk to be whichever you want the 'final version' of the file to contain. (For example, maybe "Hello world from everyone!") Then save the file.

You can then `git commit` your changes (You probably won't have to `git add` the file, because git did it for you.) and push them. Having done so, try typing `git log --graph` to see a visualization of what happened. Good job, you just resolved a conflict!

⁴The text after `>>>>>>` is the commit hash of commit A