Assignment 1 (git intro)

1.1: Your first git repository

Install git (`https://git-scm.com/`). Clone your private assignment repository. This repository should have a name of the form `UiO-INF3331/INF3331-X`, where `X` is replaced by your UiO username. First, create a `assignment1` folder in your git repository. Then add a textfile in that directory, add it to the repository and commit it. The textfile should be named `helloworld.txt` and contain the words "Hello world". Push your first commit to github.

Name of file: `helloworld.txt`
Points: 3

1.2: Getting back old versions of files

This problem will teach you how to get old versions of a file. This is useful if you make a change you later end up regretting, or for debugging purposes. Add, commit and push a new file called `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.

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 greeting.txt
```

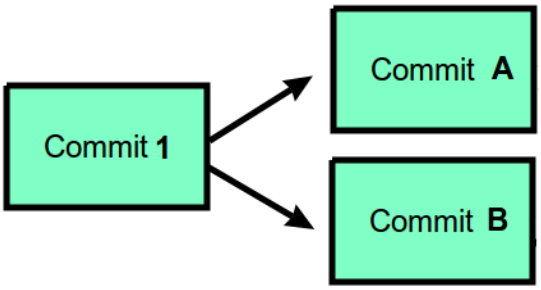
where `COMMITHASH` is the commit hash you found using `git log`. This will replace the current version of `greeting.txt` with the version from the commit you chose, and `git add` it for you. Finally, use `git commit` to make a commit which reverts the greeting to the old version.

Name of file: `greeting.txt`
Points: 2

1.3: What is a conflict and how do I resolve it? (optional)

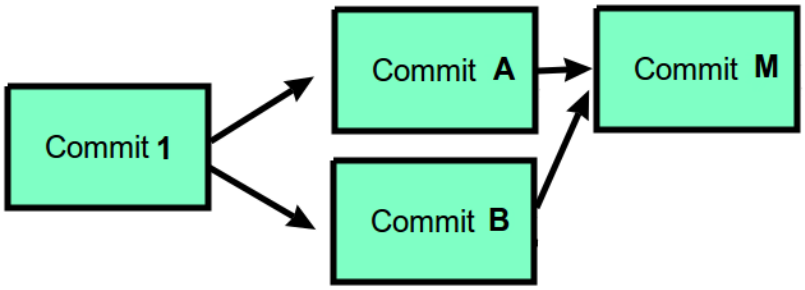
This exercise will use a lot of words to explain a common 'error' which happens when using git to work with other people. It is a good idea to read through it because it will happen even if nobody does anything wrong, and understanding the reasons why git complains makes fixing its complaints a lot less frustrating[1].

[1] This is also good life advice.



In git, each commit goes "on top of" another commit. This means that any changes made in the commit are *intentional,* so git won't bother you about them. 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 concerned[2]. It 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 commitB 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 (commit B), and so won't allow you to push[3] until you have done a `git pull` to get commit A.



When you `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 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 do it for you. Other times, maybe A introduced some new stuff you need to adapt B to work with - maybe a function you use

[2]In the real world, each person having their own *branch* would be tidier and minimize the time spent resolving conflicts, but we won't go into that here.

[3]It's possible to tell git to do the push anyway. This is generally a bad idea.

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.

To test this out, 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.

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 you will need to resolve the conflict 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 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 contain[4].

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.

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