Harvard John A. Paulson School of Engineering and Applied Sciences

F. Wermelinger
Office: Pierce 211

Homework 1

Setup Git repo, Command line, Markdown, Git

Issued:	September 1st, 2022
Due:	September 13th, 2022

Note this is a 2 week exercise. *Do not procrastinate the work.* Some additional exercises can be found in the `README` file for this exercise, see https://code.harvard.edu/CS107/main/tree/master/homework/hw1.

Note that Problem 2 is part of the pair-programming section. This assignment contains quite some text and the actual work is not in relation to the amount of pages. Pleas make sure you read through the assignment carefully as it contains some information that will be helpful to you throughout the class.

Problem 1: Class Survey (*10 points*)

Please fill in the following class survey: https://forms.gle/NKJMGQ36fPvvhei57

- ***10 points*** if the survey has been filled out and submitted before the homework deadline.

Problem 2: Setup Your Class Git Repository

In this problem you setup your class Git repository. This repository will reside inside a directory on your laptop. You are entirely free how you manage your directory structure locally. One possibility might be to have a `classes` directory below which you have directories for your classes and within them class related data including your private class repository for CS107/AC207. For example:

```
classes/
|-- CS107
|   |-- git
|   |   |-- myrepo <- your private Git repository
|   |   \-- main
|   \-- CS107_other_data
\-- CS205 (some other class in this directory)
```

The following steps are necessary for successful assignment submissions in CS107/AC207. *We will not be able to grade your work in this class if you have not performed the following steps.*

a) Sign in on https://code.harvard.edu. This is a GitHub Enterprise instance managed by Harvard University. The username you see in your profile (something similar to abc123) is your Harvard NetID.

   Material and work for this class is managed through the CS107/AC207 organization on that platform. It is located at https://code.harvard.edu/CS107 and you need to request access to it by sending your NetID to the teaching staff cs107-staff@g.harvard.edu.

   You need access to the CS107/AC207 organization in order to proceed with the next subtasks.

b) Once you have been added to the CS107/AC207 organization, proceed with the remaining steps in this tutorial: https://harvard-iacs.github.io/2022-CS107/pages/tutorials.html#tutorial-repo-steps.

   This will create your repository on the *remote*. Now you need to set it up on your laptop. You can clone the repository by repeating the following steps on other computers as well.

   i) Create the directory for your *local* repository. If you follow the suggestion above, this command could look like this

   ```
   $ mkdir -p classes/CS107/git/myrepo
   $ cd classes/CS107/git/myrepo
   ```

   The second command changes into the newly created directory.

   ii) The next step is to *initialize* a new local repository. If you have not used Git before, make sure you issue the following two commands

   ```
   $ git config --global user.name "Your Name"
   $ git config --global user.email you@harvard.edu
   ```

   As you guessed it, they will tell Git a little something about you. This information is necessary whenever you create commits such that people know who is responsible for the work in the commit. You may not necessarily use your Harvard email if you prefer to use another one instead. You can now initialize your local repository with

   ```
   $ git init
   ```

   iii) Now you can add the remote repository such that Git knows where to push changes to

```
$ git remote add origin git@code.harvard.edu:CS107/abc123.git
```

This assumes that your NetID is `abc123`. You can obtain the link to your remote from inside your repository by clicking on the green button in the top right that says "Code".[1] It is suggested you pick the SSH version but HTTPS would also work. Note that the general convention is to name the main remote repository `origin`. You can name it whatever you like but consistency is good practice when you name things (especially when you work with many Git repositories). Your local repository is now ready. You can always list the configured remotes using

```
$ git remote -v
```

At this point you can perform some customization if you like. By default, Git names the *default branch* `master` (deprecated behavior). This information will be propagated to the remote with your first push. If you like to change the name of the branch you can do so by

```
$ git branch -M main
```

to rename it `main`.[2] Note that `master` is subject to change. You can configure Git with your own name for a default branch using

```
$ git config --global init.defaultBranch <name>
```

Common names are `main`, `trunk` or `development`.

c) Before you push anything to your remote repository, you need to add some data. The correct structure of the data is defined in the syllabus of the class; see the homework submission[3] and pair-programming submission[4] sections and the linked tutorials therein.

   i) Create a directory for homework and your pair-programming labs in the root of your private repository

```
$ mkdir -p homework lab
```

   and add two `README.md` files with

```
$ echo '# My Homework' > homework/README.md
$ echo '# My Pair-programming' > lab/README.md
```

[1] A remote in Git may not necessarily be an URL, it may just as well be a path to another repository on your local file system.

[2] This is the name of the branch and is not related to the CS107/AC207 `main` repository found here https://code.harvard.edu/CS107/main

[3] https://harvard-iacs.github.io/2022-CS107/pages/syllabus.html#homework-submission

[4] https://harvard-iacs.github.io/2022-CS107/pages/syllabus.html#pp-submission

ii) It is very important to keep your history clean from polluting files. Examples of such files are object files of compiled code, editor backup files or the `.DS_Store` file on MacOSX. Such files do not contribute to the value of software and should be *ignored* in your repository. You can achieve this by setting up a `.gitignore` file. You can create this file in any (sub-)directory you like. It is customary to have a global `.gitignore` on the repository root. You can create one with the following commands

```
$ echo -e "**/__pycache__\n*.pyc\n.DS_Store\n" > .gitignore
$ echo -e "*~\n*.bak\n*.swp\n*.zip\n" >> .gitignore
$ echo -e ".bash_history\n**/.cache\n**/.local" >> .gitignore
```

You are free to add more patterns you like to ignore now or at any moment in time. Note that this will happily ignore the specified file patterns. In case you must add one of those files at some point, you can always force add them using the `-f` option in `git add -f`.
You can check the status of your repository with `git status` *(this is an important command and you should become accustomed to using it often)*.

iii) Add the *untracked* data to the *staging area* with

```
$ git add .gitignore homework lab
```

and create your first commit with

```
$ git commit -m 'Initial commit'
```

iv) Now you can issue your first push to the remote repository with

```
$ git push -u origin master
```

Depending whether you have chosen different naming, please replace `origin` (the remote) and `master` (the branch to push) accordingly. You only need the `-u` option with the two arguments whenever you push a new branch to a remote that *is not tracked* by Git.
If you have chosen to link your remote to the HTTPS URL you will need to setup a personal access token in your GitHub profile. If you have chosen the SSH URL you will need to setup a SSH key and upload the public key to GitHub. SSH keys are stored in `~/.ssh` by default. If you do not have one you can create it with

```
$ ssh-keygen -t rsa -b 4096
```

Choose the default location by just hitting enter. You may enter a password for the key or just hit enter to go without password. If go with password you will have to enter it every time you use the key. To upload the public key to GitHub, click on your icon in the top right corner on your `code.harvard.edu` page, then click on "Settings" and then "SSH and GPG keys" in the left panel. Alternatively use this link `https://code.harvard.edu/settings/keys`. Click on the green "New SSH key" button in the top right corner and give your new key a title (e. g. the name of your laptop). In the key field paste the contents of your *public key* found in `~/.ssh/id_rsa.pub`. Use for example

```
$ cat ~/.ssh/id_rsa.pub
```

and copy paste the output into the "Key" field on your GitHub page. You are now able to access any repositories on https://code.harvard.edu with corresponding permissions. Never share your private key `~/.ssh/id_rsa` with anybody.

> Do not setup this key inside the class Docker container. The key will be lost when you exit the container. You should create the key on your laptop running your native OS or in the Windows subsystem for Linux.

These steps will become second nature to you after this homework and the next few lectures. What you have done is create a homework directory using some common Unix/Linux commands, generated a `README.md` file also using a common Unix/Linux commands, and followed the usual Git workflow of staging (via `git add`), commiting changes to your local repo (via `git commit`), and pushing to the remote repository. Note that the `-m` option to `git commit` allows you to specify a commit message directly from the command line rather than via your default text editor (e.g. `vim`).

d) Homework is solved on specific Git branches and submitted via pull requests that target your default branch. See the homework tutorial https://harvard-iacs.github.io/2022-CS107/pages/tutorials.html#tutorial-hw for the details. The following steps should guide you through for this homework. You are expected to do this on your own for the following homework.

   i) Be sure you are on your default branch (e. g. `master` or `main`). You can create the required branch for this homework with

   ```
   $ git checkout -b hw1
   ```

   Alternatively if your Git version is 2.23 or newer you can use

   ```
   $ git switch -c hw1
   ```

   > We will only be grading the work you do on this branch. You will lose *5 points* if you do not solve the homework on the `hw1` branch. As a token of good will, we will not take these 5 points off for this first homework.

   You are free to make other branches off of `hw1` or your default branch, say for individual problems, but you are responsible to merge this work into your active homework branch if you want the work graded.

   ii) Create your `hw1` directory for this homework

   ```
   $ mkdir -p homework/hw1/submission
   ```

   Note that this will also create the `submission` sub-directory in which you are expected to put your homework solution for grading.
   Optional: instead of creating your `hw1` directory manually, you could add the main class repository as another remote and checkout the relevant homework material using Git

```
$ git remote add class git@code.harvard.edu:CS107/main.git
$ git fetch class
```

The name `class` for this remote is again arbitrary. The `git fetch` command updates this remote locally. You can then checkout this (or future) homework material with

```
$ git checkout class/master -- homework/hw1
```

assuming you are located at the root of your private repository. This is useful if there are additional or auxiliary files distributed with the homework, you can get it all in one go. If you perform a `git status` you will observe that new files have been added in the *staging area*. It is good practice to commit them before you continue

```
$ git commit -m 'Checkout hw1'
```

You can check your current history with

```
$ git log --oneline
96f55f8 (HEAD -> hw1) Checkout hw1
4ae4ade (origin/master, master) Initial commit
```

It tells you that there are two commits and you are currently on the `hw1` branch (`HEAD` is pointing to `hw1`).

iii) Finally, you can create a pull request at this point or wait until you are done with this homework. This step is optional at this point *but is required to submit your homework before the deadline*. You can follow the tutorial on the class website https://harvard-iacs.github.io/2022-CS107/pages/tutorials.html#tutorial-hw-example3.
Up to here, your private repository should look at least similar to

```
myrepo
|-- homework
|   |-- hw1
|   |   \-- submission
|   \-- README.md
\-- lab
    \-- README.md
```

Problem 3: Command Line (*30 points*)

Deliverables:

1. `P3.sh` script with the following lines (exact order):
   i. Command for b.)
   ii. Command for c.)
   iii. Command for d.)
   iv. Command 1 for e.)
   v. Command 2 for e.)
2. `P3f.sh` script

For the following tasks you will submit a single Unix/Linux command that outputs the desired result. Note that there are many different ways to accomplish these tasks. A short example of the absolutely essential Unix commands can be found in the `README.md` file provided with this homework.

Example question: use two Unix commands (on a single line) to count the number of lines, words and characters in the file `notes.txt`.

Example solution:

```
$ cat notes.txt | wc
      6      69     470
```

a) Due to the varying outputs across differing operating systems, it is strongly recommend that you use the CS107/AC207 Docker container to run and test your commands in a consistent environment. *Grading will be based on this environment.*

You can follow these steps and ask your TF for help should you get stuck.

1. Install Docker on your machine following https://docs.docker.com/get-docker
2. Ensure the Docker application is running
3. Open a command line terminal
4. Pull the CS107/AC207 class container with

```
$ docker pull iacs/cs107_ubuntu
```

5. You can run the container using the provided run script in the class repository https://code.harvard.edu/CS107/main/blob/master/docker/run_cs107_docker.sh
6. Type `exit` in the Docker container window to leave the container.

Assume your directory looks like this (you can copy the `run_cs107_docker.sh` in your private repository and commit it for example)

```
.
|-- homework
|   |-- hw1
|   |   |-- apollo13.txt
|   |   \-- submission
|   |-- run_cs107_docker.sh
|   \-- README.md
|-- lab
|   \-- README.md
\-- run_cs107_docker.sh
```

You can then launch the Docker container with

```
$ ./run_cs107_docker.sh homework/hw1
root@e7fd83a9233a:~# ls
apollo13.txt  submission
```

Note that this has automatically mounted your `hw1` directory with the necessary data for this exercise.

b) ***5 points***

The `apollo13.txt` file is provided in the `code/p3` directory of the homework handout. Write a command that counts the number of lines in the file that contain one or more numerical digits and write the result to an output file called `out.txt`. Your command should assume that the text file is in the current directory.

For example, if the provided text file contained:

```
a9
2c
abd
939
```

Then the resulting command should produce the following file.

```
$ cat out.txt
3
```

There are multiple ways to accomplish this count. Attempts should involve the `grep` command. Two possibilities are

```
$ grep -c [0-9] apollo13.txt > out.txt
$ grep [0-9] apollo13.txt | wc -l > out.txt
```

The correct count is 14213 lines.

c) ***5 points***

Write a command using `grep` that outputs a *single line* that tells what the `--count` (or `-c`) option does in `grep`.

Hint: *Use* `grep --help` *for a nicely formatted output of options.*

Hint: *Your output should be a single line.*

Hint: *MacOSX is based on a BSD Unix flavor for which the* `grep` *interface is implemented slightly different. On MacOSX there is no* `grep --help` *option. You may use* `man grep` *instead which is slightly more difficult to obtain a single line output. Alternatively, you can use the class Docker container.*

The most straightforward solution is

```
$ grep --help | grep '\--count'
```

d) ***5 points***

Write a command that counts the total number of `.py` files in your *current directory*. For example, if the current directory contains the following:

```
file1.py
directory/
file2.c
file3.py
```

then the output of the command should be 2.

Note that no files are provided in the homework handout. If you want to play around, feel free to create some files to test things. An easy way to create an empty file is to use the command `touch file.py` for example.

The most straightforward solution is

```
$ ls *.py | wc -l
```

e) ***5 points***

Write a command that counts the number of:

1. files in the current directory and sub-directories that "others" **do not** have *read and write access* to.
2. files and directories in the current directory that "others" **do not** have *read and write access* to.

Neglect hidden files and directories (for example `.file`).

For example, if the current directory contains the following:

```
.
|-- -rw-rw-rw- file1
|-- -rw------- file2
|-- -rw-rw-r-- .hidden_file
|-- drwxrwxrwx subdir1
|   \-- -rw-rw-rw- file3
```

```
\-- drwxr-xr-x subdir2
     |-- -rw-r--r-- file4
     \-- -rw-r--r-- file5
```

then the output should be 3 for item 1 and 2 for item 2 above.
Item 1 can be solved with the `find` command

```
$ find . -type f ! -path "*/.*" ! -perm -o=rw | wc -l
```

Item 2 can be solved with the `find` command as well

```
$ find . -maxdepth 1 -name "[!.]*" \( -type f -or -type d \) \
    ! -perm -o=rw | wc -l
```

Note that the option grouping using "`\(`" and "`\)`" to enforce the logical or for file or directory is important. See the `OPERATORS` section in `man find`. The backslashes are needed to *escape* the parenthesis which would be interpreted by the shell instead if they are missing. The solution contains a tar archive with file and directory permissions as in the example given in the problem formulation above. Use

```
$ tar -xzpf e.tar.gz
```

to extract the directory (the `-p` option preserves the permissions).

f) ***10 points***
Write a bash script that counts and prints the number of *non-empty lines* of each file in *the current directory* along with its name. Your resulting file should be called `P3f.sh`. Empty lines may include white space.
For example, if the current directory contains the following:

```
file1.py
directory/
file2.c
file3.py
```

then the output should look like

```
file1.py 5
file2.c 8
file3.py 2
```

if the individual files had the given number of *non-empty* source lines.
A simple solution would involve a `for`-loop and `sed` to delete empty lines:

```
for f in $(find . -maxdepth 1 -type f); do
    echo "${f##*/} $(cat $f | sed -r '/^\s*$/d' | wc -l)"
done
```

Additional spaces in the output format are neglected. Transposed line count and file names are accepted and do not deduce points.

Problem 4: Markdown (*10 points*)

Deliverables:

1. `P4.md`

The task in this problem is to write a Markdown document the file `P4.md`. Markdown is a very easy and powerful language to work with. An overview of the basic language features can be found in the link below.

https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet

The task is to write a short Markdown document. Your Markdown document must have at least all of the following components:

- Two different types of headers
- Text with bold emphasis
- A List
- A table
- A link

Optional: a powerful tool in combination with a Markdown document (and many other formats as well) is https://pandoc.org. It allows you to quickly convert a simple Markdown document into a PDF (using LaTeX as a backend) or HTML document. The following example converts the `P4.md` document into `P4.pdf`

```
$ pandoc -o P4.pdf P4.md
```

Please see `solution/code/p4` for a valid solution.

Problem 5: Git (*50 points*)

Deliverables: for this problem, copy the `hw1/sandbox` directory from the forked repository discussed below into your `hw1/submission` directory in your private Git repository. The following contents should be in the `sandbox` directory:

1. `README.md`
2. `P5_fork.png`
3. `P5_merge.png`
4. `P5_remote.png`

You must also create a *separate* pull request for this problem as described at the very end of the exercise.

An important part of software development is being able to control and revise a *history of changes* made to the source from possibly different contributors. Version Control Systems (VCS) are the primary tools used to keep track of actual changes to code. Git is the most common VCS used and has established to an industry standard. It is what we will use for this course.

See the `solution/code/p5/P5.sh` script for a standalone replay of tasks 5a to 5b.

a) ***10 points***

In this problem we are working with another Git repository that can be found in the class organization `https://code.harvard.edu/CS107/sandbox`. You can clone this repository using the `git clone <URL>` command. You can use the URL provided above or visit the website in your browser and click on the green "Code" button in the top right to learn about other options.

It is important that you do not clone this repository inside your private Git repository or any other Git repository for that matter. This would create *nested* repositories which you want to avoid. If you have followed the suggested directory structure mentioned earlier, you could change into the `git` directory and issue the `git clone` command in there to arrive at something that looks like the following

```
classes/
|-- CS107
|   |-- git
|   |   |-- myrepo <- your private Git repository
|   |   |-- sandbox <- the cloned sandbox for this problem
|   |   \-- main
|   \-- CS107_other_data
\-- CS205 (some other class in this directory)
```

Cloning the repository will automatically *checkout* the default branch and you will be able to see the files associated with that branch locally. The problem is that you only have read access for this repository and you can therefore only clone or pull from it

but not contribute your own changes via push for example. There are two options to resolve this:

1. Have the repository owner give you push access. This is often not desired if you are not a main contributor or developer for the project.
2. Create your own *fork* of the repository and make a pull request (PR) back to the original repository when you have made all the changes. This is the approach that you will take in this assignment. It is the workflow used on platforms like GitHub and you are using it to submit your homework as well. The difference for your homework is that you create the PR *within your own repository*, while in practice PR's are *between* repositories (either branches with the same name or differently named branches). You should be familiar with this workflow.

In order to proceed with a *fork* of `https://code.harvard.edu/CS107/sandbox` you must remove your clone from before first. Visit the URL above and click on the "fork" button in the top right corner. Fork the repository to your own account and create a screenshot called `P5_fork.png` of the landing page (you should see "forked from CS107/sandbox" in the top left corner.

Clone your *forked* repository in the same manner as you did before for the read-only `CS107/sandbox` repository (which you have deleted again). Save the screenshot in `homework/hw1/sandbox` in the forked repository, add it using `git add`, commit it using `git commit -m "Add forked screenshot"` and push the commit with `git push`.

b) ***25 points***

One of the most fundamental concepts in Git is the concept of a branch. A branch is simply a sequence of commits, where the branch is referenced by the most recent commit in this sequence. Branches *can therefore change their reference* over time, while the reference for a commit is *immutable*.

Branches are the place where you develop and throw things around. You should never do that in your default branch which is usually accessible to the public. Creation of branches is *cheap* in Git and at its core philosophy.

i) You can check a list of branches with

```
$ git branch -a
```

You can see there is a `test` branch in the forked repository. You can switch branches in Git using the

```
$ git checkout # works also in Git 2.23 and later
$ git switch   # only for Git 2.23 and later
```

Checkout the `test` branch. You can inspect the *differences* on this branch using the command

```
$ git diff master..HEAD
```

where here `master` is the default branch (the one you were before) and `HEAD` is a shorthand for the currently checked out commit (you could also omit it in the command above). Spend some time figuring out what the differences are, you can also use other commands to navigate around and list files, for example `ls`. It is perfectly valid that branches may contain very different files or even have files missing. Most often branches will look somewhat similar except for the features they implement.

ii) Next you will create your own branch. First switch back to the default branch with

```
$ git switch master
```

We did this step because we want to create the new branch based off of the default branch. Naturally, this can be any valid branch.
Name the new branch "`hw1p5`". You can create the branch with

```
$ git branch hw1p5
```

and switch to it with

```
$ git switch hw1p5
```

Because this is such a common procedure, the two operations can be combined with either

```
$ git checkout -b hw1p5
```

or

```
$ git switch -c hw1p5
```

Create one more branch named `hw1p5_tmp` and switch to it.

iii) You should now be on the branch `hw1p5_tmp`. Change into the `homework/hw1/sandbox` directory and edit the `README.md` file. Add your name at the very bottom of the file. Save and close the file. You can use your text editor of choice.
To check modifications on the current branch you use the `git status` command. *This is a very important command and you should use it often for orientation*. After your modifications you should see

```
$ git status # executed on repository root
On branch hw1p5_tmp
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   homework/hw1/sandbox/README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

The changes have not been *staged* yet. This process adds the changes that you want to *commit* next to the so called *index*. You can *add* changes to the index using the `git add` command. Add the changes you made in `README.md` to the index. After this operation `git status` should report

```
$ git status # executed on repository root
On branch hw1p5_tmp
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   homework/hw1/sandbox/README.md
```

You can continue to make changes to files (including `README.md`) and repeat the process if you want it to be committed next. Only the currently staged changes will be committed. Which is what you do now. Type `git commit`.
This will open up the default editor (`vim` on most systems) to edit the *commit message*. You can set the `EDITOR` (globally) or `GIT_EDITOR` (only for Git) environment variables if you want to change the editor for future sessions. You are most likely in `vim` now. To enter your commit message type "`i`" to enter *insert mode*. Now you can type your commit message. Press the "`esc`" key to enter *normal mode* again where you can save and exit by typing "`:wq`" (`:` is to enter *command mode*, `w` is for *write* and `q` is for *quit*).
So far we have used `git commit -m 'Commit message'` to bypass the editor if the commit message *is only small*, i. e., small changes. In general, your commit messages must be concise and contain enough information to describe the changes in the commit. Writing is generally a hard task, so is writing good commit messages.
If you now type `git status` again you will see that there are no other changes and you can proceed with the next logical unit that should eventually become a commit. Note that you can also add *partial* changes in a file to the index. See the `--patch` option in `git help add`.

iv) You now have a local commit that has not been pushed to the remote repository of your sandbox fork. You can accomplish this with `git push`. This command will fail because you are working on the `hw1p5_tmp` branch and Git does not know what to do with it because you have not specified enough argument to the push command. You must be more specific by telling Git which *remote* and which *branch* you want to push

```
$ git push origin hw1p5_tmp
```

If you know you will push frequently on this branch because it is a hot development, add the `-u` option such that Git will remember what to do when you are on this branch. If you add this option you can just use `git push` later on. If you want to know how local branches relate to remote branches you can do so with

```
$ git branch -vv
```

v) Recall what you did up to this point. You have modified the `README.md` file by adding your name at the bottom. We now simulate something that can not be avoided in distributed version control and therefore must be expected when working with Git. Although Git is good at resolving this issue autonomously, it will ask for your help when it gets stuck.
Switch back to your `hw1p5` branch that you have created earlier. Open again the `README.md` file. You will notice that your name has disappeared from the file because you have not added it on this *branch*. Now update this `README.md` file in *the same place* where you added your name before (at the bottom of the file). Write something creative other than your name, then add and commit these changes similar to before (you do not need to push).
At some point in the development you want to *merge* branches together again (e. g. merge a working feature into the default branch). For this you use the `git merge` command. We now want to merge our changes on the `hw1p5_tmp` into our current branch (`hw1p5`). To do this type

```
$ git merge hw1p5_tmp
```

This merge will result in a merge conflict because the two `README.md` files contain *different* changes at the same place and Git does not know which it should favor. You should see the following error

```
$ git merge hw1p5_tmp
Auto-merging homework/hw1/sandbox/README.md
CONFLICT (content): Merge conflict in homework/hw1/sandbox/README.md
Automatic merge failed; fix conflicts and then commit the result.
```

Check the status with `git status`. You have to resolve the conflict manually in this case. Open the conflicting `README.md` file in your editor and take a screenshot. Save the screenshot in the same directory as the `README.md` file and name it `P5_merge.png`. You should see the following

```
# CS107 / AC207 Homework 1 Sandbox

Fall 2022: Fork, pull requests, merge conflicts
<<<<<<< HEAD

This text will cause a conflict
||||||| e55df4c
=======

Fabian
>>>>>>> hw1p5_tmp
```

Git has marked the conflicting sections. The markers mean the following

- "`<<<<<<< HEAD`" marks the beginning of the conflicting section in the *current branch* (`HEAD`)
- "`||||||| e55df4c`" marks the end of the conflicting section in current branch and marks the beginning of the content in the *common ancestor* of the two branches that are merged. The reference of the common ancestor is further indicated.
- "`=======`" marks the end of the content in the common ancestor and the beginning of the conflicting section in the branch that is being *merged in*. Note that the content in the common ancestor is empty because none of the code we added on either branch existed (we appended at the end of the file).
- "`>>>>>>> hw1p5_tmp`" marks the end of the conflicting section on the other branch.

By manually resolving a merge conflict you must read and understand the changes on both sides and possibly contact the author of either code to further consolidate. You are free to edit this file. You can pick either change or write something completely new. For example

```
# CS107 / AC207 Homework 1 Sandbox

Fall 2022: Fork, pull requests, merge conflicts

OK, the merge conflict is resolved now.
```

Save the changes and exit the editor. To finalize the merge conflict you still need to *add* your resolution to the index (check with `git status`).

```
$ git add homework/hw1/sandbox/README.md
```

and finally type `git commit` to conclude the merge. You will notice that Git has automatically added a commit message for you. You can use this suggestion or edit further for more content. Save and exit the editor. The merge conflict is now resolved. After merging a branch, it is good practice to *delete* the branch as it is now fully contained in the target branch. You can use `git branch -d hw1p5_tmp` to do this. You can delete the *remote* branch using `git push origin --delete hw1p5_tmp`.
You can also add and commit the `P5_merge.png` file if you have not done so yet. Push the changes to the remote.

vi) You should still be on the `hw1p5` branch. Make one more change to the `README.md` file (e. g. add another line of Markdown) then add, commit and push the change. Since commits are *immutable* you cannot undo that commit.
You can however *revert* a commit using the `git revert` command. In order to do this you need the reference to the commit (its SHA1 hash). You can find this information using `git log` for example. The hash is a number of the form `d037e425`. You only need as many digits as necessary for Git to uniquely identify

the commit. Since we want to revert the most recent commit, we could also use `HEAD` as a shortcut. You can inspect where `HEAD` is pointing to with

```
$ cat .git/HEAD
ref: refs/heads/hw1p5
$ cat .git/refs/heads/hw1p5
d037e425b44d779331fd86c23ba423520b649d3c
```

To revert the previous commit (that was already pushed to the remote) use

```
$ git revert HEAD
```

This will create a new commit with the changes undone. Check `git log` to inspect the new commit. Finally push the commit to the remote.
Note that reverting a commit necessarily creates a new commit in the history because commits are *immutable*. This was necessary in this case because the commit we reverted has *already been pushed to the remote*. This means the history is shared with other developers working with this remote. If you had not pushed, the commit would still be *local* to the history of the repository on your laptop. In that case you can actually remove the commit such that the "deletion" is not visible in the history using `git reset`. This process is also related to an *interactive rebase* which will be discussed further in the lecture.

c) ***15 points***

In the previous section we were working with branches. Because we have forked the repository, there is a relation ship between the fork and the original repository. These are *two different* repositories, however, and a relationship between the two has to be established through *remotes*. With this, you can update a local branch using commits on a possibly different branch maintained on the other remote (i. e. in the other repository).

You also want to maintain this relationship to ensure a smooth merging process when you request to pull a branch from your fork into the original repository. You could follow a similar strategy between your private Git repository and the main class repository, where new material is posted. This would allow to easily update your private repository with new material from the other remote.

In this task we want to add a new remote that points to the original sandbox repository in the CS107 organization, `https://code.harvard.edu/CS107/sandbox`. Since this repository is read-only, you can only fetch/pull from it. You can check your currently configured remotes with

```
$ git remote -v
origin  git@code.harvard.edu:<your NetID>/sandbox.git (fetch)
origin  git@code.harvard.edu:<your NetID>/sandbox.git (push)
```

You can add a new remote called `upstream` (you can name it whatever you like, `upstream` is often used) invoke the following command

```
$ git remote add upstream https://code.harvard.edu/CS107/sandbox
```

If you check your remotes you can verify that it was indeed added

```
$ git remote -v
origin  git@code.harvard.edu:<your NetID>/sandbox.git (fetch)
origin  git@code.harvard.edu:<your NetID>/sandbox.git (push)
upstream        https://code.harvard.edu/CS107/sandbox (fetch)
upstream        https://code.harvard.edu/CS107/sandbox (push)
```

Take a screenshot of this terminal output and name it `P5_remote.png`. Save the screenshot in `homework/hw1/sandbox` while still on the `hw1p5` branch. Commit and push this addition.

You have now configured two remotes `origin` (obtained by cloning the forked repository) and `upstream` which points to the original repository. You can push and pull from the former but only pull from the latter. In order to push to `upstream` you must create a pull request. As discussed earlier, you can specify the remote and branch to `git push` and `git pull` to specify a specific destination or source.

You are almost done! The final step is to create a pull request to merge your `hw1p5` branch from your forked repository into the `master` branch in the original repository `https://code.harvard.edu/CS107/sandbox`. You can achieve this on the webpage of your forked repository. Make sure your PR contains all the steps we went through in this problem. If you did it correct, you can see your PR appear in the `https://code.harvard.edu/CS107/sandbox` repository.

For completeness, copy your `homework/hw1/sandbox` directory from this problem into the `homework/hw1/submission` directory in your *private Git repository* for which you must issue another PR as described in the class syllabus.