Lecture 02 - Version Control with Git (last updated: 10/27/22)

Git is a version control system designed for source code and other kinds of plain text. It is great for synchronizing work when you are working in a team, as well as for keeping track of changes you make when you are working alone.

In this class, you are required to use Git for your homework assignments (labs). You use it to obtain skeleton code, track your progress, and submit your work. You can also use Git to view my solutions, once I release them.

Git is first and foremost a command-line tool, invoked using the "git" command. "git" actually encompasses a number of subcommands, such as "git clone", "git pull", and "git checkout". You can see the man pages for each subcommand using the --help flag:

```
$ git clone --help  # brings up the man pages for git clone

$ man git-clone     # does the same as above
```

Git command cheat sheet

There are plenty of hands-on guides teaching you how to use Git. Here are a few we recommend for getting started:

- man gittutorial: the official Git tutorial included in its man pages
- git - the simple guide (https://rogerdudler.github.io/git-guide/): a guide that gets straight to the point with nice illustrations
- Yet Another Git Guide (https://j-hui.com/pages/yagg/): my own Git guide that I shares with my collaborators

Here is a summary of the Git commands you should know for this class:

- git status: show the status of the current working directory
- git log: view commit history
- git add: add changed files to the staging area
- git commit: create a commit from changes checked into the staging area
- git push: push commits to a remote repo
- git pull: pull changes from a remote repo
- git clone: obtain a copy of another repo with its commit history

Anatomy of a repository

Git organizes collections of files as repositories (aka "repos"):

- A repo is always "rooted" in some directory; all "tracked" files must live within that directory tree.
- Repos keep track of the version history of their files in a "commit log" (AKA "commit history").
- Each Git repo contains a special, hidden .git/ directory where the commit history and other metadata is stored.

You can run Git commands from any subdirectory within a repo.

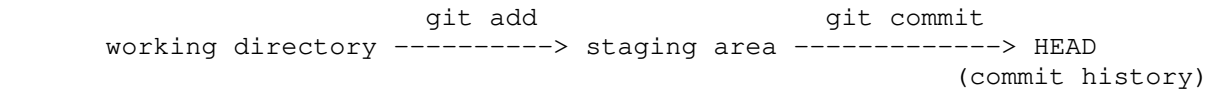
From working directory to commit

Git manages multiple copies of your repo's contents:

- the working directory is what you can see and edit, i.e., outside of .git/
- the staging area (AKA "index") is what will form the next commit
- each commit has a snapshot of your repo's contents at some point in time

Thus, there are also multiple copies of each (tracked) file in your repo.
A file is tracked if it is in the staging area or in some commit. When you git add a file, you copy it from your working directory to the staging area.

The most recent commit in your repo is known as HEAD. When you git commit, you create a new commit (now the new HEAD) from the contents of your staging area.



Git uses some clever tricks to make sure that identical copies of the same file don't take up unnecessary space (beyond the scope of this lecture).

Note that when you clone a repo, you will only receive its commit history; modifications in the working directory and staging area are not cloned.

The Git file life cycle

git status will tell you when there are differences between the copies of a file in the working directory, the staging area, and HEAD. This is helpful for keeping track of which modifications end up in your next commit. The output of git status usually looks something like this:

```
$ git status

Changes to be committed:
        modified:   foo.txt

Changes not staged for commit:
        modified:   bar.txt

Untracked files:
        baz.txt
```

These sections are to be interpreted as follows:

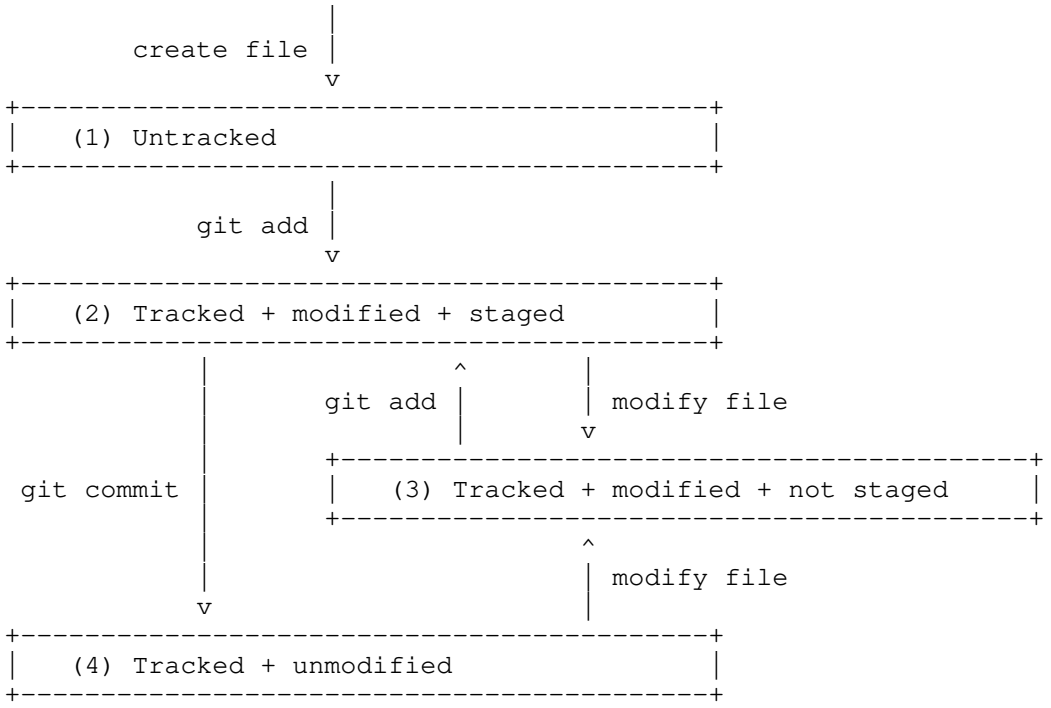
(1) If a file appears in the "Untracked files" section like baz.txt, then it only exists in your working directory, and is not yet tracked in the staging area or commit history.

(2) If a file appears in the "Changes to be committed" section like foo.txt, then the staged copy of that file differs from the copy in HEAD.

(3) If a file appears in the "Changes not staged for commit" section like bar.txt, then the working copy of that file differs from its staged copy.

(4) If a file does not appear in any of these sections, then the working, staged, and HEAD copies of that file are identical.

It is sometimes helpful to think of these statuses as stages in the life cycle of each file tracked by Git:



Note that it is possible for a file to have both staged (2) and unstaged (3) changes, i.e., if it has been modified since it was git added. The life cycle diagram does not illustrate this scenario.

.gitignore files (optional)

There are often files that you don't ever want to track in a repo. For example, it is considered bad practice to track object files and executables (you will learn about these in a later lecture). It can be cumbersome to see them in your git status, and it's easy to accidentally track these files.

You can use a .gitignore file to tell Git that you don't want to track a file. For example, this .gitignore tells Git not to track any files named "a.out", or whose file name ends with ".o":

```
$ cat .gitignore
a.out
*.o
```

Any files matching those rules will not show up in git status, nor will they be tracked when you git add them. These .gitignore rules only apply to directory the .gitignore file is in, and any of its subdirectories. You can place different .gitignore files in different directories to ignore files more selectively. You can read the man pages (man gitignore) for more usage details.

It is considered good practice to include .gitignore files in your Git repos. You should also track .gitignore files themselves in your repo, since the same .gitignore rules will likely apply to any others who clone your repo.

Anatomy of a commit

We refer to each commit by its "commit hash", which is a long string of characters computed from the following information:

- the file contents of the commit
- the commit message (e.g., git commit -m "the commit message")
- the author and timestamp of the commit
- its parent commit(s), i.e., the previous commit(s) in the commit history

If any of these change, the commit hash will also change.

When you git commit, the old HEAD will become the parent commit of the new HEAD. For this course, we will only work with repos with a linear commit history: where each commit has a single parent, and where each parent has one child.

It is possible for a commit to have multiple children if different commits are made with the same parent. You will often encounter such divergent commit histories when using Git to collaborate with others. You have two options in order to join a divergent history:

- create a merge commit, i.e., a commit with multiple parent commits
- rebase one set of commits onto another, modifying their commit hashes

When you do either of these, Git will try to automatically merge the different sets of changes, but will stop and ask you to resolve merge conflicts if it encounters any.

Other useful Git commands

- git ls-files: list tracked files
- git mv/rm: rename or remove a tracked file
- git diff: see changes between the working directory and staging area
- git diff --cached: see changes between the staging area and HEAD
- git diff HEAD: see changes between the working directory and HEAD
- git restore: restore file in working directory from staging area
- git restore --staged: restore file in staging area from HEAD