Select the right page in Gradescope or we will not grade the question!

Homework 2
Git

Due: Wednesday, September 25th, 11:59PM (Hard Deadline)

1 Git's Chicken and Egg Problem

Git has a (well-deserved) reputation as a tool with a high learning curve. There are several reasons for this. One is that git has a bad user interface – the commands and flags you use to get things done are frustratingly inconsistent and simply must be learned over time. The bigger reason, however, is that using git requires having an understanding of how git works and learning how git works requires having some experience using git. This circular dependency makes it hard to get started as there is no good way to break in. The best we can do is try to hold your hand as you get started.

Complete at least 4 of the scenarios of the course at `https://katacoda.com/courses/git`.

As you complete these scenarios, the progress bar on the home page should start to fill up. Feel free to do more levels if you want!

Q1: Whenever you're finished, add a screenshot of the page where the progress bar is visible.

Some other tutorials / resources

There is a *lot* written on the Internet about how to learn git. Some highlights:

- For a deeper dive into git from the basics to the very advanced, check out Atlassian's tutorials (Atlassian is the company behind BitBucket and JIRA).
- For more visual learners, check out this interactive visual learning game – caveat this game simplifies some things, but I think it's still a useful tool to learn from. There's even some low-hanging fruit lying around to improve this tool.
- A lighter, more humorous "quick-reference" guide and a more, uh, frustrated version (warning: language).

2 Using git in your own projects

This sets up the Winter 2015 EECS 280 P2 as an example. 280 students please don't get confused!!

0. Install git: `sudo apt install git`

Using version control well is a habit learned over time. As a first step, for the rest of the term, the **first** thing you should do whenever an EECS class assigns you a project is:

1. Create a repository

```
> mkdir -p eecs280-w15/project2
> cd eecs280-w15/project2
> git init
```

2. Grab a copy of the spec

```
> wget 'https://drive.google.com/uc?id=0B4qlH840ZwikRmQtdkRIeTZjODA&export=download' -O spec.pdf
```

3. Grab any starter code

```
> wget 'https://drive.google.com/uc?id=0B4qlH840ZwikbkZLS3Z5YTVSeW8&export=download' -O eecs280-w15-p2.tgz
> tar -xf eecs280-w15-p2.tgz
> rm eecs280-w15-p2.tgz
```

4. Create a blank starter cpp file

```
> touch p2.cpp
```

5. Add it all to git and commit it

```
> git add *
> git status # Check to make sure everything looks right
> git commit
```

Do this **before** even beginning to read the spec. Make it a habit.

If you try typing `make`, you'll find that this starter code does not compile. The spec has a great tip at the bottom of page 11 that suggests adding stub functions. (No, you don't actually need to read this spec)

There are 15 functions we need to stub out. Fortunately, your c4cs staff have saved you some trouble. You should download our copy of `p2.cpp`:

```
> wget 'https://raw.githubusercontent.com/the-alex/c4cs-eecs280/master/p2.cpp' -O p2.cpp
```

Now try running `make`. Cool; the project builds. But what changed? Fortunately, git can help us figure this out.

Q2: What command will show you the difference between your working directory and already committed changes?

Now, go ahead and commit the changes to `p2.cpp` only.

You ran `git status` before committing those changes, right? (Because it's *always* a good idea to run `git status` *all* the time). You probably noticed that annoying "Untracked Files" section that lists all of the compiled output the Makefile made. It doesn't really make sense to track compiled output in a version control system (it is hard to track changes on a binary file such as an executable).[1] Instead, we would prefer if git would ignore the built output.

Commit a change so that built output is ignored. When you are done, `git status` should print

```
On branch master
nothing to commit, working directory clean
```

Q3: Once you have completed all of the previous steps, copy the output of:

> **`git log -n2 -p | head -n 40`**

2.1 Preparing for the future

This question summarizes the setup and simple usage loop for git. Make some changes, add them, and commit them. If you need to delete a file, use `git rm file_to_delete` instead of just `rm`. From this moment onward, you should use git for all of your EECS projects. Seriously. Use it.

Set up git repositories for all of your other EECS projects this term. If you are currently in the middle of a project, create an initial commit at its exact state right now (even though it is not yet finished / not working).

* * *

Later in the term we will analyze how you have been using git and how you may improve. It is important that you have at least one non-trivial project (e.g. a class project) that you have used git to maintain.

* * *

Repositories from group projects are fine. If you use a different version control system (`svn`, `bzr`, `hg`, `darcs`, `cvs`, `rcs`, `...`), choose one project to manage with git instead, it's worth learning, you will encounter it.

[1] Some binary files are good to check in though, such as the project specification, which will rarely or never change.

3 A little about shells and your *environment*

Try the following:

```
# The /tmp directory holds temporary files. It's a great place to do some
# quick tests or experiments, however it is automatically emptied every time
# the machine reboots. This means if your machine crashes, you *will* lose
# everything in /tmp.
#
# Don't ever put anything you care about in /tmp !!

> mkdir /tmp/throwaway
> cd /tmp/throwaway
> git init
> echo hello > world
> git add world
> git commit
```

Git automatically opens a text editor for you to enter a commit message.

Q4: What text editor did git open for you? _____

Git is not the only program that will open an editor automatically. If you don't like the default choice, we can change it using an *environment variable* called `EDITOR`. Try the following:

```
> echo "it's a small" > world
> git add world
> export EDITOR="emacs -nw"
> git commit
```

Note: Shells are fussy about spaces, ***do not*** *put spaces around the = in these commands*

Q5: What text editor did git open for you? _____

Q6: What does the `-nw` flag do? (Hint: Try it without `-nw`)

Perhaps, however, you do not want to change the editor used for every program, but only for git. We can do that too. Try the following:

```
> echo "on top of the" > world
> GIT_EDITOR=gedit git commit world
```

Q7: If you set `EDITOR` and `GIT_EDITOR` to different values, which takes priority?

Some extra things to think about (i.e. not graded): Notice that we used two different approaches for setting environment variables. In the second example, we set the environment variable in-line, which means it only lasts for the lifetime of that command. If you run `git commit` again without the `GIT_EDITOR=` part, git will use the old editor.

In the first case we used the `export` command, which sets that variable permanently for the lifetime *of that shell*. If you make another commit in that shell, git will use the same value of `EDITOR`. What happens if you open a new terminal? Which editor gets used?

It would be annoying to manually set `EDITOR` every time you open a new shell. When `bash` (the shell program) starts, it reads the file `~/.bashrc`, that is the "hidden file" (leading `.`) named `.bashrc` in your home directory (`~` is a shortcut to your home). Try adding `export EDITOR="gedit"` to your `~/.bashrc` file and then make some new commits. Does the editor in your *current* shell change? What about if you open a *new* shell? What if you don't include the "`export`"?