

Homework 2

Git

Due: Saturday, September 24, 10:00PM (Hard Deadline)

Submission Instructions

Submit this assignment on [Gradescope](#). You may find the free online tool [PDFescape](#) helpful to edit and fill out this PDF. You may also print, handwrite, and scan this assignment.

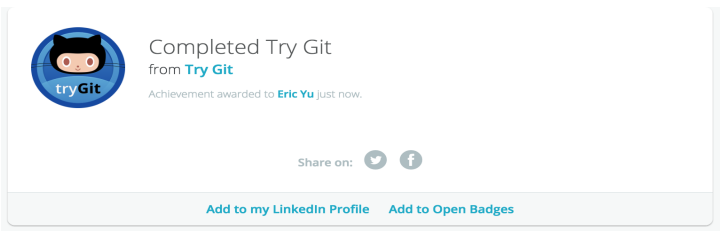
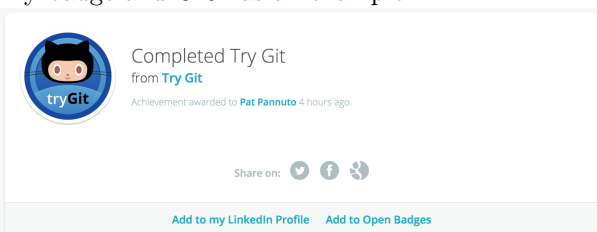
1 Git's Chicken and Egg Problem

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.

Code Academy and GitHub teamed up to build a nice, online git beginners tool. As you work through the exercises, pay attention to the explanations of each step at the top of the page, and the “Advice” section at the bottom/bottom-right of the page. The text introduces several new pieces of git-specific language.

Complete the online course at <https://try.github.io>. Grab a screenshot of your badge and copy the URL here. (You will need to sign in or create an account get your badge)

My badge and URL as an example:



<https://www.codeschool.com/users/2276474/badges/121>

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 eeecs280-w15/project2
> cd $_
> 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 eeecs280-w15-p2.tgz
> tar -xf eeecs280-w15-p2.tgz
> rm $_
```

4. 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. That could be a pain, but fortunately the header file has already done most of the work. We'll drill into this command in the future, but for today, simply run:

```
> echo -e '#include <cassert>\n#include "p2.h"\n' > p2.cpp
> grep ';' p2.h | grep -v ' \*' >> p2.cpp
```

People can sometimes have strong opinions about text editors. One reason is that the editors that many programmers use can do far more than simply edit text. Let's look at one example. Start by opening `p2.cpp` with `vim`:

```
> vim p2.cpp
# press 'jjj'      so that your cursor is on the i of int sum(...)
# press 'qq'      to begin recording a macro into the vim register q
# press 'f;'      jump to the ';' character
# press 's'       remove character under the cursor and enter insert mode
# insert the needed text:
# ' <space>{<enter>assert(false);<enter>}<enter><escape>'
# press 'j'       so that your cursor is on the i of int product(...)
# press 'q'       to finish recording the macro
# press '16@q'    16 times play the commands stored in register q
# save and quit vim
```

Now try running `make`. Cool, a building project in 60 seconds or less.

Commit the changes.

Note: This example happened to use `vim`, other editors also supports macros, for example in `emacs` `C-x (`, `C-x)`, and `C-x e` will record, finish, and execute a macro.

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
```

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

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

```
commit 3b1388580a8505f16dac125fd8ac5497e9a00869
Author: Eric Yu <echenyu@umich.edu>
Date:   Fri Sep 16 01:47:44 2016 -0400

    ey - fixed some spacing on bitvector

diff --git a/hw1/BitVector.h b/hw1/BitVector.h
index f9f5aef..1b9ff15 100644
--- a/hw1/BitVector.h
+++ b/hw1/BitVector.h
@@ -7,10 +7,10 @@ using namespace std;

class BitVector {
public:
-
-    BitVector(string const &bitString_in) : bitString(bitString_in) {}
-    size_t size() { return bitString.size(); }
-    bool operator[](const int index) { return bitString[index] == '0' ? 0 :
1; }
+    string get_string() const { return bitString; }
+
+    void push_back(bool val) {
+        ostringstream o;
@@ -19,14 +19,9 @@ public:
    }

    friend ostream &operator<<( ostream &output, const BitVector &bVector ){
-
-    output << bVector.bitString;
-    return output;
-    }
-
-    string get_string() const {
-        return bitString;
+        output << bVector.bitString;
+        return output;
+    }
-    }
-
-    private:
-        string bitString;
-    };
@@ -84,7 +79,7 @@ BitVector &operator^(const BitVector &a, const BitVector &b)
{
33d
```

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.

¹Some 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.

What text editor did git open for you? Git opened Vim for me

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*

What text editor did git open for you? This time, git opened emacs

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

The nw flag runs emacs in the terminal instead of opening a new window. (However, it seems to do something different for vim. There was a help description message when using the nw flag with vim.

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
```

If you set **EDITOR** and **GIT_EDITOR** to different values, which takes priority?

By following the steps from above, the GIT_EDITOR would take priority. When I set EDITOR to vim, and GIT_EDITOR to emacs, git would open emacs instead of vim.

Roughly how long did you spend on this assignment? Maybe around an hour??

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”?