Getting started with Git

This semester we'll be using git to distribute and collect programming assignments. Some of you have used git in other classes, while others are probably new to it. Below are some instructions to help get you started using Git at IU. These just scratch the surface of git's features; please also check out the (many) online tutorials for more information.

1. Setting up SSH keys

If you haven't used git on the CS linux machines (e.g. burrow) before, you'll have to do the following steps.

1. Open a web browser and go to `http://github.iu.edu/`.
2. Click Login, then enter your IU username and password.

   If you've used git on burrow before, you can stop here and go on to Step 6.
3. Check if you have a file called `~/.ssh/id_rsa.pub`. If not, type `ssh-keygen` at the Linux prompt, and just press Enter in response to all the questions.
4. Copy the contents of the `~/.ssh/id_rsa.pub` to your local (e.g. Windows) clipboard.
5. In the github website, click Settings, then SSH keys. Paste the text you copied in the previous step into the box and press Submit.
6. Now, return to the main Github page (click on the little cat icon in the upper left), and then click on the green New Repository button.
7. Create a new repository called b551-test. Click the Private option, and check the Initialize this repository with a README. Then press the green Create Repository button.
8. Now, at the Unix command line, type:

   ```
   git clone git@github.iu.edu:youruserid/b551-test.git
   ```
9. You should now see a directory called `b551-test` in your current directory, and within that a file called README. If so, git is working! If not, **please** double check that you've done the above steps correctly, and/or ask an AI for help before continuing.

2. Basic git commands and concepts

Git is a system for helping you as you write your programs, by tracking the changes you make to your program over time. In the process, it can also help you back up your code, so that (if you use it correctly!) it's virtually impossible to ever accidentally lose your work. It also makes collaborating with other people (like your teammate!) much easier. It has a learning curve but, in the long run, will save you time if you learn to use it well. (Trust us!)

In some sense, Git is like a fancy version of a cloud storage system like Google Drive or Dropbox. You work on your program like normal, saving it to the disk as you go. However from time to time, when you've done some amount of work that feels like a milestone (like implementing a new feature, fixing a bug, or stopping your coding for the evening), you issue git commands to commit a new version of your code and push it to the cloud. Git tracks the changes you make, so if you ever wonder in the future what changed between this morning's version and this evening's version, it's easy to see. If you ever make a mistake, like accidentally deleting your code or introducing a bug that you can't seem to find, Git makes it easy to retrieve the old version. Let's try it out now.

1. Once you completed the steps in section 1, you created a Git storage area (or repository) for testing purposes, and already cloned it on burrow. Go into your `b551-test` directory using `cd`.

2. Create a new file, like a small Python program that prints Hello World. Once you have it working, it's a nice moment to commit to git. To do this, we'll use a series of three commands:

   ```
   git add hello.py
   git commit -m "adding hello world program"
   git push
   ```

   The message in quotes is a description of the changes you've made to your code since the last time you committed it to git. You'll always want to include a descriptive comment so that later you can find the right version, if needed. You'll almost always use the above three commands in sequence: `add` to tell git that you've made an update to a particular file, `commit` to mark a new version and add a comment, and `push` to send it to the cloud.

3. Now, open a web browser and go to `http://github.iu.edu/`. You should see the lab3 repository in your list – try clicking on it. If all has worked so far, you should be able to click on `hello.py` to browse the source file you just committed.

4. Let's say we want to make a small change to the program. Back in burrow, edit the program so that it prints out the sum of 490 and 659. Once you have that working, push the change to git again by running the three commands in Step 2. Remember to put a nice descriptive message inside the quotes, like "added gratuitous sum."

5. Now back in the web browser, click on the `hello.py` file again. You should see the newest version with your recent updates. But what happened to the old version? It's still there, too! Click on the History tab, and you'll see a list of all of the versions of this program – there should be three so far, including the initial one. You can click on the Browse code link next to any version to see the code exactly as it was at the time you committed, or you can click on the numbers and letters (something like "f83c900" – this is called the SHA code and is just a unique version number) to see exactly what changed between this version and the previous version.

We'll see many other useful features of git as we go on, but that's what you need to get started. The more often you commit, the better — I try to commit at least every 15-20 minutes of coding, or whenever I feel like I've accomplished something. Commiting to git is also the way you'll submit assignments, so always make sure to commit the final version when you're done. We've set things up so that the instructors will be able to see the code you submit, but no one else will.

3. Working on teams

For the assignments, we (the course instructors) will automatically make repositories for groups and fill them with skeleton code. For example, if Jingya (wang203) and David (djcran) were a team for assignment one, the repository would be something like **cs-b551/djcran-wang203-a1**. The basic workflow for using git with a team is the same as in section 2 above, but there are the added complications that multiple of you may be editing files at the same time. Here's how it works.

1. You can use the **git clone** command to set up the new repository on burrow from github.iu.edu.

2. Use **git add** and **git commit**, like before, whenever you have a change you want to save. This usually means that you've fixed a bug, or added a small feature. No need to be stingy on commits you can do them as often as you want, and generally more often is better than less often. Changes you commit are not visible to your partner.

3. When you have your code in a place where you want to share it with your partner, run **git add**, **git commit**, and then also **git push**. This pushes your changes to the github server so that your partner can see them. Typically you only want to push once your code is in a sort of stable state – i.e. it runs without syntax errors and it's something you want them to be able to see. Your partner will now be able to see the changes by using the github website, but your changes won't be visible on their copy in burrow automatically. That's because they've probably made changes to their version of the code at the same time you've been making changes to your version.

4. When your partner is ready to download your changes into their copy of burrow, they should **git add** and **git commit** to save any local changes to the files they've made. Then they run **git pull**, which downloads your changes and attempts to combine your changes with any changes they've made. At this point, a couple of things could happen:

   (a) If your partner hasn't changed the code at all, then things are very easy. They now have your changes locally and can continue working.

   (b) If your partner made some changes, then git will try to automatically merge your changes together with their changes. It's usually very good at doing this, you'll just see a message asking you for a commit comment, and you can just enter something like "combining together our changes".

   (c) If your partner made changes to the same lines of code that you changed, you've created a conflict which you'll have to resolve manually. Git will alert you of this:

   Automatic merge failed; fix conflicts and then commit the result.

   (d) Now if you open the program file in a text editor, you'll see some extra lines that git has added, that look like <<<<<<<, >>>>>>>, and ==========. These lines mark your version of the code, and your partners' version of the code. Your job now is to replace these lines with the version of code you actually want i.e. by merging your changes and your partner's changes together. Once you've done this, simply git add, git commit, and git push.

That's it! So in summary, the basic workflow is to add and commit regularly, then push when you want to share changes with your partner and pull when you want to incorporate your partner's changes in your code. You can avoid problems with conflicts by deciding ahead of time who will work on which part of the code, to avoid both changing the same part of the code at the same time.