**Assignment 1: Introduction Lecture**

**Learning Objectives**

1. Understand how to set up a githup account
2. Understand the multi-user workflow in git version control
3. Understand how to resolve conflicts in git version control

**Part one: Git Distributed Version Control**

(Note: No submission is required for this part. However, I will be able to verify on github that you have completed this section)

**1. Introduction**

Git is a version control system (VS) also called a source code management (SCM) tool, which was created for a single task: managing changes to a programmer’s files. It lets you track every changes of a software project goes through, as well as where those changes came from. This makes Git an essential tool for managing large software projects. In this course, we will be using Git as our revision control and source code management system. We will be using GitHub for centralized online repository hosting. This will enable us to adopt an agile software development methodology so your group can rapidly collaborate and iterate on the design, verification, and evaluation of the lab assignments and projects. This tutorial covers how to: setup your GitHub account, use Git for basic incremental development, use GitHub to collaborate with your group, manage Git branches and GitHub pull requests. Git commands are part of linux of commands. References to Linux commands if needed are listed below:

* Summary of linux commands (posted on BB)
* Linux Command Line form free-electrons (posted on BB)
* Useful tutorials at http://www.ee.surrey.ac.uk/Teaching/Unix/

This assignment was adapted and modified from an initial git tutorial written by Prof. Christopher Batten of Cornell University.

**2. Setting up your GitHub Account**

GitHub is an online service that hosts centralized Git repositories for a growing number of opensource projects. It has many useful features including a web-based source browser, history browser, branch management, merge requests, code review, issue tracking, and even a built-in wiki attached to every repository. We have created a dedicated GitHub organization for the course located here:

• <https://github.com/lewisu-research-compsci>

For most of this tutorial you will be using a repository in your own personal GitHub account. If you do not yet have an GitHub account, you can create one here:

• <https://github.com/join>

Be sure to use your Lewis University email address. Once your account is setup, please make sure you set your full name so we can know who you are on GitHub. Please also consider uploading a profile photo to GitHub; it makes it more fun to interact on GitHub if we all know what each other look like. Go to the following page and enter your first and last name in the Name field, and then consider uploading a profile photo.

• <https://github.com/settings/profile>

Once your account is created sent your GitHub ID via email to [ngalamlu@lewisu.edu](mailto:ngalamlu@lewisu.edu). If you are doing this assignment on your personal computer you to install “git bash” for windows (<https://git-scm.com/download/win>).

Configure git by running the following commands on git bash:

**$ git init**

**$ git config -- global user.name “Your Name”**

**$ git config -- global user.email your.email@example.com**

Before you can begin using GitHub, you need to create an SSH key pair and upload the corresponding SSH public key to GitHub. GitHub uses these keys for authentication. The following link provides information about creating an SSH key pair which you can use:

<https://help.github.com/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/>

To test things out try the following on git bash

**$ ssh -T** [**git@github.com**](mailto:git@github.com)

You may see a warning about the authenticity of the host. Don’t worry, this is supposed to happen the first time you access GitHub using your new key. Just enter “yes”. The GitHub server should output some text including your GitHub ID. Verify that the GitHub ID is correct, and then you should be all set. There are two good GitHub Guides you might want to take a look at:

• <https://guides.github.com/activities/hello-world>

• <https://guides.github.com/introduction/flow>

GitHub has two integrated tools that students might find useful: an issue tracker and a wiki. Consider using the GitHub issue tracker to track bugs you find in your code or to manage tasks required to complete the lab assignment. You can label issues, comment on issues, and attach them to commits. See the following links for more information about GitHub issues:

• <https://guides.github.com/features/issues>

• <https://help.github.com/articles/about-issues>

Consider using the GitHub per-repository wiki to create task lists, brainstorm design ideas, rapidly

collaborate on text for the lab report, or keep useful command/code snippets. See the following links

for more information about GitHub wikis:

• https://guides.github.com/features/wikis

• <https://help.github.com/articles/about-github-wikis>

**3. Git and Github**

In this section, we begin with a basic single-user workflow before demonstrating how Git and Github can be used for effective collaboration among multiple users. We discuss how to resolve conflicts and how to manage branches and pull requests.

**3.1. Single-UserWorkflow**

In this section, we cover some basic Git commands and illustrate a simple Git workflow. We have created a Git repository that we will be using as an initial template, so the first step is to fork this tutorial repository. Forking is the process of making a personal copy of someone else’s repository on GitHub.

A local repository is a first-class mirror of the remote repository with the entire history of the repository, and thus almost all operations are essentially local requiring no communication with GitHub. The following commands create a subdirectory for this tutorial in your home directory before using the git clone command to clone the remote repository and thus create a local repository.

**$ mkdir -p ${HOME}/cpsc36000**

**$ cd ${HOME}/ cpsc36000**

**$ git clone git@github.com:** [**lewisu-research-compsci**](https://github.com/Lewisu-cpsc36000)**/**[**assign0-git**](https://github.com/Lewisu-cpsc36000/assign0-git) **tu2**

**$ cd tu2**

**$ TUTROOT=${PWD}**

The git clone command takes two command line arguments. The first argument specifies the remote repository on GitHub you would like to clone, and the second argument specifies the name to give to the new local repository. Note that we created an environment variable with the directory path to the local repository to simplify navigating the file system in the rest of this tutorial. You should never directly manipulate anything within the .git directory.

**$ cd ${TUTROOT}**

**$ ls -la**

Let’s assume we want to create a new file that contains a list of fruits, and that we want to manage

this file using Git version control. First, we create the new file.

**$ cd ${TUTROOT}**

**$ echo "apple" > fruit.txt**

To manage a file using Git, we need to first use the git add command to tell Git that it should track this file from now on. We can then use git commit to commit our changes to this file into the repository, and git log to confirm the result.

**$ cd ${TUTROOT}**

**$ git add fruit.txt**

**$ git commit -m "initial fruit list"**

**$ git log**

The -m command line option with the git commit command enables you to specify a commit message that describes this commit. All commit messages should include a "subject line" which is a single short line briefly describing the commit. Many commits will just include a subject line (e.g., the above commit). If you want to include more information in your commit message then skip the –m command line option and Git will launch your default editor. You still want to include a subject line at the top of your commit message, but now you can include more information separated from the subject line by a blank line.

Note, you can learn about any Git command and its usage by typing git help <command>, where

<command> should be substituted by the actual name of the command. Students are encouraged to

learn more about each Git command beyond the details covered in this tutorial.

The git log command displays information about the commit history. The beginning of the output

from git log should look something like this:

**commit 0e5b2b2c05b5837839554fa047e52e121c8206b1**

**Author: cb535 <cb535@lewisu.edu>**

**Date: Sat Aug 18 18:01:17 2015 -0400**

**initial import of fruit**

Conceptually, we should think of each commit as a copy of all of the tracked files in the project at the time of the commit. This commit just included changes to one file, but as we add more files each commit will include more and more information. The history of a git repository is just a long sequence of commits that track how the files in the repository have evolved over time. Notice that Git has recorded the name of who made the commit, the date and time of the commit, and the log message. The first line is the commit id which uniquely identifies this commit. Git does not use monotonically increasing revision numbers like other version control systems, but instead uses a 40- digit SHA1 hash as the commit id. This is a hash of all the files included as part of this commit (not just the changes). We can refer to a commit by the full hash or by just the first few digits as long as we provide enough digits to unambiguously reference the commit. Now let’s add a fruit to our list and commit the change.

**$ cd ${TUTROOT}**

**$ echo "mango" >> fruit.txt**

**$ git commit -m "added mango to fruit list"**

Unfortunately, this doesn’t work. The output from git commit indicates that there have been no changes since the last commit so there is no need to create a new commit. Git has a concept of an index which is different compared to other version control systems. We must “stage” files (really we stage content not files) into the index, and then git commit will commit that content into the repository. We can see this with the git status command.

**$ cd ${TUTROOT}**

**$ git status**

which should show that fruit.txt is modified but not added to the index. We stage files in the index with git add like this:

**$ cd ${TUTROOT}**

**$ git add fruit.txt**

**$ git status**

Now git status should show that the file is modified and also added to the index. Our commit

should now complete correctly.

**$ cd ${TUTROOT}**

**$ git commit -m "added mango to fruit list"**

**$ git status**

So even though Git is tracking fruit.txt and knows it has changed, we still must explicitly add the files we want to commit. There is a short cut which uses the -a command line option with the git commit command. This command line option tells Git to commit any file which has changed and was previously added to the repository.

**$ cd ${TUTROOT}**

**$ echo "orange" >> fruit.txt**

**$ git commit -a -m "added orange to fruit list"**

**$ git status**

Staging files is a useful way to preview what we will commit before we actually do the commit. This helps when we have many changes in our working directory but we don’t want to commit them all at once. Instead we might want to break them into smaller, more meaningful commits or we might want to keep working on some of the modified files while committing others.

Figure 1 illustrates how the commands we have used so far create a single-user development workflow. The git clone command copies the remote repository to create a local repository which includes both the working directory and the special .git directory. The git add command adds files to the index from the working directory. The git commit command moves files from the index into the special .git directory. The -a command line option with the git commit command can commit files directly from the working directory to the special .git directory. Now that we have made some changes, we can use git log to view the history of last few commits and then add another line to the fruit.txt file.



Figure 1: Git and Github for Single-User Development

**$ cd ${TUTROOT}**

**$ git log**

**$ echo "plum" >> fruit.txt**

**$ cat fruit.txt**

Imagine you didn’t like your changes and want to revert the changes, you would use the git checkout command as below.

**$ cd ${TUTROOT}**

**$ git checkout fruit.txt**

**$ cat fruit.txt**

As illustrated in Figure 1, the git checkout command resets any a file or directory to the state it was in at the time of the last commit. The output from the git status command should look something like this:

**$ cd ${TUTROOT}**

**$ git status**

**On branch master**

**Your branch is ahead of ’origin/master’ by 3 commits.**

**(use "git push" to publish your local commits)**

**nothing to commit, working directory clean**

The git status command is telling us that the local clone of the repository now has more commits than the remote repository on GitHub. If you visit the GitHub page for this repository you will not see any changes. This is a critical difference from other centralized version control systems. In Git, when we use the git commit command it only commits these changes to your local repository. If we have done some local work that we are happy with, we can push these changes to the remote repository on GitHub using the git push command.

**$ cd ${TUTROOT}**

**$ git push**

**$ git status**

Notice that the output of the git status command indicates that our local repository is up-to-date with the remote repository on GitHub. Figure 1 shows visually the idea that the git push command moves commits from your local repository to the remote repository on GitHub. Visit the GitHub page to verify that our new commits have been pushed to the remote repository:

• <https://github.com/lewisu-research-compsci/assign0-git>

Click on commits at the top of the GitHub page to view the log of commits. You can browse who made each commit, what changed in each commit, and the state of the repository at each commit. Return to the main GitHub page for the repository and click on the fruit.txt file to view it.

**3.2. Multi-user Workflow**

Since your tutorial repository is public on GitHub, any other user can also clone this repository. If you would like to collaborate with another GitHub user, you would need to give that user read/write permission. To emulate how collaboration with GitHub works, we will “pretend” to be different users by cloning extra copies of the tutorial repository.

**$ cd ${HOME}/cpsc36000**

**$ git clone git@github.com: Lewisu-research-compsci/assign0-git tut2-alice**

**$ cd tut2-alice**

**$ ALICE=${PWD}**

**$ cd ${HOME}/cpsc36000**

**$ git clone git@github.com: Lewisu-research-compsci/assign0-git tut2-bob**

**$ cd tut2-bob**

**$ BOB=${PWD}**

We can now emulate different users by simply working in these different local repositories: when we work in ALICE we will be acting as the user Alice, and when we work in BOB we will be acting as the user Bob. Figure 2 illustrates a multi-user development environment: both Alice and Bob have their own separate local repositories (including their own working directories, index, and special .git directories), yet they will

Let’s have Alice add another entry to the fruit.txt file, commit her changes to her local repository,

and then push those commits to the remote repository on GitHub:

**$ cd ${ALICE}**

**$ echo "banana" >> fruit.txt**

**$ git commit -a -m "ALICE: added banana to fruit list"**

**$ git log --oneline**

**$ git push**

**$ cat fruit.txt**

****

**Figure 3: Git and Github for Multi-User Development**

If you view the GitHub page for this repository it will appear that you are the one making the commit (remember we are just pretending to be Alice), which is why we used ALICE: as a prefix in the commit message.

Now let’s assume Bob wants to retrieve the changes that Alice just made to the repository. Bob can use the git pull command to pull all new commits from the remote repository into his local repository. The git pull command performs two actions, it first fetches all the updates and then merges or applies them to the local project. If there are no conflicts in the file contents, the command executes successfully. If there are conflicts, the command does not merge all the changes and reports the conflicting content. We will learn how to resolve conflicts in Section 3.3.

**$ cd ${BOB}**

**$ git pull**

**$ git log --oneline**

**$ cat fruit.txt**

Figure 2 shows visually the idea that the git pull command moves commits from the remote repository on GitHub to your local repository. Bob’s copy of tutorial repository should contain Alice’s most recent commit and his copy of the fruits.txt file should include “banana”. Now let’s assume Bob also wants to make some changes and push those changes to the remote repository on GitHub:

**$ cd ${BOB}**

**$ echo "peach" >> fruit.txt**

**$ git commit -a -m "BOB: added peach to fruit list"**

**$ git log --oneline**

**$ git push**

**$ cat fruit.txt**

Similar to before, Alice can now retrieve the changes that Bob just made to the repository using the git pull command.

**$ cd ${ALICE}**

**$ git pull**

**$ git log --oneline**

**$ cat fruit.txt**

This process is at the key to collaborating via GitHub. Each student works locally on his or her part of the lab assignment and periodically pushes/pulls commits to synchronize with the remote repository on GitHub.

**3.3. Resolving Conflicts**

Of course the real challenge occurs when both Alice and Bob modify content at the same time. There are two possible scenarios: Alice and Bob modify different content such that it is possible to combine their commits without issue, or Alice and Bob have modified the exact same content resulting in a conflict. We will address how to resolve both scenarios.

Let us assume that Alice wants to add lemon to the list and Bob would like to create a new file named vegetables.txt. Alice would go ahead and first pull from the central repository to grab any new commits from the remote repository on GitHub. On seeing that there are no new commits, she edits the file, commits, and pushes this new commit.

**$ cd ${ALICE}**

**$ git pull**

**$ echo "lemon" >> fruit.txt**

**$ git commit -a -m "ALICE: added lemon to fruit list"**

**$ git push**

Since Bob recently pulled from the remote repository on GitHub, let’s say he assumes that there have been no new commits. He would then go ahead and create his new file, commit, and attempt to push this new commit.

**$ cd ${BOB}**

**$ echo "spinach" > vegetables.txt**

**$ echo "broccoli" >> vegetables.txt**

**$ echo "turnip" >> vegetables.txt**

**$ git add vegetables.txt**

**$ git commit -m "BOB: initial vegetable list"**

**$ git push**

**To git@github.com:lewisu-research-compsci/assign0-git**

**! [rejected] master -> master (fetch first)**

**error: failed to push some refs to ’git@github.com:lewisu-research-compsci/assign0-git’**

**hint: Updates were rejected because the remote contains work that you do**

**hint: not have locally. This is usually caused by another repository pushing**

**hint: to the same ref. You may want to first integrate the remote changes**

**hint: (e.g., ’git pull ...’) before pushing again.**

**hint: See the ’Note about fast-forwards’ in ’git push --help’ for details.**

On executing the sequence of commands above, you should notice that Git does not allow Bob to push his changes to the central repository as the version of the central repository has been updated by Alice. You should see a message similar to the one above. Git suggests us to merge the remote commits before pushing the local commits. We can do so by first using the git pull command to merge the local commits.

**$ cd ${BOB}**

**$ git pull**

Git will launch your default text editor because we need to merge your local commits and the remote commits. You will need to enter a commit message, although usually the default message provided by Git is fine. After saving (Pres **ESC** and type **“:wq**” follow by the **ENTER**) the commit message, we can take a look at the Git history using git log to see what happened.

**$ cd ${BOB}**

**$ git log --oneline --graph**

**\* 656965a Merge branch ’master’ of github.com:lewisu-research-compsci/assign0-git**

**|\**

**| \* 9d943f3 ALICE: added lemon to fruit list**

**\* | 4788dbe BOB: initial vegetable list**

**|/**

**\* c7cc31e BOB: added peach to fruit list**

**\* dc28bc9 ALICE: added banana to fruit list**

The --graph command line option with the git log command will display a visual graph of the commit history. You can see that Bob and Alice worked on two different commits at the same time. Alice worked on commit 9d943f3 (your value might be different) while Bob was working on commit 4788dbe (your value might be different). Bob then merged these two sets of commits using a new commit 656965a (your value might be different). Bob can now push his changes to the remote repository in GitHub.

**$ cd ${BOB}**

**$ git push**

GitHub has a nice commit history viewer which shows a similar commit graph as we saw above:

• <https://github.com/lewisu-research-compsci/assign0-git/network>

While this approach is perfectly reasonable, it can lead to a very non-linear and complex commit history. We strongly recommend that you use an alternative called rebasing. While merging “merges” sets of commits together, rebasing will apply one set of commits first and then apply a second set of commits on top of the first set of commits. This results in a more traditional linear commit history.

Let’s reconstruct a similar situation as before where Alice adds another fruit and Bob adds another vegetable at the same time.

**$ cd ${ALICE}**

**$ git pull**

**$ echo "plum" >> fruit.txt**

**$ git commit -a -m "ALICE: added plum to fruit list"**

**$ git push**

**$ cd ${BOB}**

**$ echo "potato" >> vegetables.txt**

**$ git commit -a -m "BOB: added potato to vegetable list"**

**$ git push**

**To git@github.com:Lewisu-research-compsci/assign0-git**

**! [rejected] master -> master (fetch first)**

To rebase, we will use the --rebase command line option with the git pull command.

**$ cd ${BOB}**

**$ git pull --rebase**

**$ git push**

**$ git log --oneline --graph**

**\* 0d5fba5 BOB: added potato to vegetable list**

**\* e56ad1b ALICE: added plum to fruit list**

**\* 656965a Merge branch ’master’ of github.com:lewisu-research-compsci/assign0-git**

**|\**

**| \* 9d943f3 ALICE: added lemon to fruit list**

**\* | 4788dbe BOB: initial vegetable list**

**|/**

**\* c7cc31e BOB: added peach to fruit list**

**\* dc28bc9 ALICE: added banana to fruit list**

Study the output from the git log command carefully. Notice how instead of creating a new merge commit as before, rebasing has applied Alice’s commit e56ad1b (your value might be different) first and then applied Bob’s new commit 0d5fba5 (your value might be different) on top of Alice’s commit. Rebasing keeps the git history clean and linear.

Sometimes Alice and Bob are editing the exact same lines in the exact same file. In this case, Git does

not really know how to resolve this conflict. It does not know how to merge or rebase the two sets of commits to create a consistent view of the repository. The user will have to manually resolve the conflict. Let’s explore what happens when Alice and Bob want to add a new fruit to the fruits.txt file at the exact same time. First, Alice adds kiwi and pushes her updates to the remote repository on GitHub.

**$ cd ${ALICE}**

**$ git pull**

**$ echo "kiwi" >> fruit.txt**

**$ git commit -a -m "ALICE: added kiwi to fruit list"**

**$ git push**

Now Bob adds date and tries to push his update to the remote repository on GitHub.

**$ cd ${BOB}**

**$ echo "date" >> fruit.txt**

**$ git commit -a -m "BOB: added date to fruit list"**

**$ git push**

**To git@github.com:Lewisu-research-compsci/assign0-git**

**! [rejected] master -> master (fetch first)**

As before, Bob uses the --rebase command line option with the git pull command to pull the commits from the remote repository on GitHub.

**$ cd ${BOB}**

**$ git pull –rebase**

**First, rewinding head to replay your work on top of it...**

**Applying: BOB: added date to fruit list**

**Using index info to reconstruct a base tree...**

**M fruit.txt**

**Falling back to patching base and 3-way merge...**

**Auto-merging fruit.txt**

**CONFLICT (content): Merge conflict in fruit.txt**

**Failed to merge in the changes.**

Git indicates that it was not able to complete the rebase. There is a conflict in the fruit.txt file. We

can also use the git status command to see which files have conflicts. They will be marked as both

modified:

**$ cd ${BOB}**

**$ git status**

Git instructs Bob to first resolve the conflict and then use the --continue command line option with

the git rebase command to finish the rebase. If you take a look at the fruit.txt file you will see

that it now includes conflict markers showing exactly where the conflict occurred.

**$ cd ${BOB}**

**$ cat fruit.txt**

**Apple**

**mango**

**orange**

**banana**

**peach**

**lemon**

**plum**

**<<<<<<< 7f8ec0fb9d7c705e3caf9034f3b96b0c8e7cad92**

**kiwi**

**=======**

**date**

**>>>>>>> BOB: added date to fruit list**

This shows that the commit from the remote repository on GitHub has kiwi as the last line in the file, while the last commit to the local repository has date on the last line in the file. To resolve the conflict we can directly edit this file so that it reflects how we want to merge. We can choose one fruit over the other, choose to include neither fruit, or choose to include both fruit. Edit the file using your favorite text editor to remove the lines with markers <<<<, === and >>>> so that the file includes both fruit.

**$ cd ${BOB}**

**$ cat fruit.txt**

**apple**

**mango**

**orange**

**banana**

**peach**

**lemon**

**plum**

**kiwi**

**date**

The next step is critical and easy to forget. We need to use the git add command to add any files that we have fixed! In this case we need to use the git add command for the fruit.txt file.

**$ cd ${BOB}**

**$ git status**

**$ git add fruit.txt**

**$ git status**

Notice how the output from the git status command has changed to indicate that we have fixed the conflict in the fruit.txt file. Since we have resolved all conflicts, we can now continue the rebase:

**$ cd ${BOB}**

**$ git rebase --continue**

**$ git push**

**$ git log --oneline --graph**

**$ cat fruit.txt**

Resolving conflicts is tedious, so to avoid conflicts you should communicate with your group members which student is going to be executing which files. Try to avoid having multiple students working on the same file at the same time, or at least avoid having multiple students working on the same lines of the same file at the same time.

**3.4. Branches and Pull Request**

In this section, we describe branches and pull requests which are slightly more advanced topics but tremendously useful. Students could probably skim this section initially, and then revisit this information later in the semester. Branches and pull requests enable different students to work on different aspects at the project at the same time while keeping their commits separated in the remote repository on GitHub. So far, all of our work has been on the master branch. The master branch is the primary default branch. Creating additional branches can enable one student to work on a new feature while also fixing bugs on the master branch, or branches can enable students to experiment with some more advanced ideas but easily revert back to the “stable” master branch.

Let’s say that Alice wants to work on a new list of animals in Alice and Bob’s shared repository, but she wants to keep her work separate from the primary work they are focusing on. Alice can create a branch called alice-animals and commit her new ideas on that branch. It is usually good practice to prefix branch names with your Github ID to ensure that branch names are unique. The following commands will first display the branches in the local repository using the git branch command before creating a new branch called alice-animals.

**$ cd ${ALICE}**

**$ git branch**

**$ git checkout -b alice-animals**

**$ git branch**

**$ git status**

The git branch command uses an asterisk (\*) to indicate the current branch. The git status command also indicates the current branch. Alice can now create a new file and commit her changes to this new branch.

**$ cd ${ALICE}**

**$ git branch**

**$ echo "cow" > animals.txt**

**$ echo "pig" >> animals.txt**

**$ echo "dog" >> animals.txt**

**$ git add animals.txt**

**$ git commit -m "ALICE: initial animal list"**

**$ git log --oneline --graph --decorate**

The --decorate command line option with the git log command will show which commits are on which branch. It should be clear that the alice-animals branch is one commit ahead of the master branch. Pushing this branch to the remote repository on GitHub requires a slightly more complicated syntax. We need to specify which branch to push to which remote repository:

**$ cd ${ALICE}**

**$ git push -u origin alice-animals**

**$ cat animals.txt**

The name origin refers to the remote repository that the local repository was originally cloned from (i.e., the remote repository on GitHub). You can now see this new branch on GitHub here:

• <https://github.com/lewisu-research-compsci/assign0-git/branches>

You can browse the commits and source code in the alice-animals just like the master branch. If Bob wants to checkout Alice’s new branch, he needs to use a slightly different syntax.

**$ cd ${BOB}**

**$ git pull --rebase**

**$ git checkout --track origin/alice-animals**

**$ git branch**

**$ cat animals.txt**

Alice and Bob can switch back to the master branch using the git checkout command.

**$ cd ${ALICE}**

**$ git checkout master**

**$ git branch**

**$ ls**

**$ cd ${BOB}**

**$ git checkout master**

**$ git branch**

**$ ls**

The git branch command should indicate that both Alice and Bob are now on the master branch, and there should no longer be an animals.txt file in the working directory. One strength of Git is that it makes it very easy to switch back and forth between branches. Once Alice has worked on her new branch, she might be ready to merge that branch back into the master branch so it becomes part of the primary project. GitHub has a nice feature called pull requests that simply this process. To create a pull request, Alice would first go to the branch page on GitHub for this repository.

• <https://github.com/lewisu-research-compsci/assign0-git/branches>

She then just needs to click on New pull request next to her branch. You must carefully select the base fork! If you simply choose the default you will try to merge your branch into the repository that is part of the Lewisu-research-compsci GitHub organization. Click on base fork and select Lewisu-research-compsci/assign0-git. Alice can leave a comment about what this new branch does. Other students can use the pull request page on GitHub to comment on and monitor the new branch.

• <https://github.com/lewisu-research-compsci/assign0-git/pull/1>

Users can continue to develop and work on the branch until it is ready to be merged into master. When the pull request is ready to be accepted, a user simply clicks on Merge pull request on the GitHub pull request page. When this is finished the Git history for this example would look like this:

**$ cd ${ALICE}**

**$ git pull**

**$ git log --oneline –graph**

**\* f77c7f2 Merge pull request #1 from laime/alice-animals**

**|\**

**| \* fe471e9 ALICE: initial animal list**

**\* | 80765f3 BOB: added date to fruit list**

**|/**

**\* 7393cac ALICE: added kiwi to fruit list**

**\* 4c1fff6 ALICE: added kiwi to fruit list**

**\* 1982bea BOB: added potato to vegetable list**

**\* 5fd4d2e ALICE: added plum to fruit list**

**\* eaab790 Merge branch ’master’ of github.com:laime/** **assign0-git**

**|\**

**| \* e44ab18 ALICE: added lemon to fruit list**

**\* | 4a053a6 BOB: initial vegetable list**

**|/**

**\* 2a431b0 BOB: added peach to fruit list**

**\* 98ad45a ALICE: added banana to fruit list**

**Part two: General knowledge about Computer Science**

**Problem 1.**

The reading materials assigned for the first week of lecture cover three themes that are: Algorithm analysis, algorithm correctness, and distributed algorithms. Write a half-page abstract that summarizes these themes.

**Problem 2**

Slides #1 and 2 of the lecture presentation list the hottest topics in computer science. In one page describe which topics apply to the activities of your company/profession and identify some of the problems that you think a research-based approach can be used to solve.

## **Submission**

Initially hw3.txt should be pulled from the github hw3 repository. You must submit the updated version of hw3.txt to the github hw3 repository. To do so, follow these instructions after you have opened Linux:

1. $ **mkdir hw3**
2. $ **cd hw3**
3. $ **cp** [Your hw3 text file location] **hw3.txt** # replace the braces with the location of your hw3.txt file
4. $ **cd ..**
5. $ **git add hw3/hw3.txt**
6. $ **git commit -m "Homework 3 submission"**
7. $ **git tag -f "hw3"** # The tag MUST be "hw3". Failure to do so will result in loss of credit.
8. $ **git push origin master --tags** # Note the "--tags" at the end. This pushes tags to github

Using tags allows the course instructor to determine which commit represents your hw3 submission, even if you make later commits to your repository.