Lab 4: Git and Debugging

Pre-Lab

- **Don't** pull from the skeleton repository yet. Pulling from the skeleton repository will cause a merge conflict, which you will learn how to fix later in lab. Even if you already know how to fix merge conflicts, we want you to fix it in a very specific way, so please do not be tempted to pull yet!
- If you've already pulled from skeleton, please contact your TA for how to proceed with the lab.
- This lab requires that you got a full score on `lab1` on GS. If you haven't please contact your TA before proceeding.
- In this lab, we will be working with git. Before we do that, let's make sure your repository is synced up with GitHub. To do this, simply run:

```
git push origin master
```

If the command succeeds, you're good to go! If not, you likely got an error message like this one. Please follow the instructions there for how to fix this error.

Introduction

Since this is midterm week, this lab will be shorter than usual. In this lab, you will take a deeper dive into git and also practice debugging one more time with a cool exercise. By the end of this lab, you should feel much stronger about the git workflow, and your debugging skills should be more refined!

Git Background

In this part, you will watch a series of videos that explain the major concepts of git, and then get some hands-on practice using git. Please watch all six videos below. Itai has been known to talk slowly, so feel free to increase the playback speed if necessary!

- Git Intro - Part 1
- Git Intro - Part 2
- Git Intro - Part 3
- Git Intro - Part 4
- Git Intro - Part 5
- Git Intro - Part 6

In particular, after watching the video, you should understand the following concepts:

- Local git workflow: `git add` and `git commit`
- Moving between commits and updating files with `git checkout`
- Detached `HEAD` states
- Remote repositories, e.g. those hosted on GitHub
- Local git integration with remote repositories: `origin/master` and `skeleton/master`
- How to resolve merge conflicts

If you have any questions about the above concepts, you can reference Sarah’s Git Guide, Git WTFS (Git Weird Technical Failure Scenarios – get your mind out of the gutter!), or you can always feel free to ask your TA to clarify concepts.

We will warn you here: *be wary of git information that you find online*, since not all of it comes from trusted sources. Also, be sure to **NEVER copy commands you find online or get from friends with git when you are stuck – always ask your TA when you are in doubt**.

Git Exercise

Now, it's time to practice what you've learned. Remember, this lab requires that you have gotten a full score on the Lab 1 Autograder on Gradescope. If you have not completed Lab 1, please consult with a TA. Whenever you're ready, please pull the starter code using the following command, but **don't freak out when you get a merge conflict!** Please read on for more information.

```
git pull skeleton master
```

We've purposely introduced a merge conflict in your `lab1` directory. The goal of this exercise is to have you practice some git concepts including merge conflicts, detached head states, and checking out files in your `sp21-s***` repo. By the end of this exercise, you should feel more comfortable maneuvering the commit tree on the commandline to keep your files in a desired state.

The merge conflict in the `lab1` directory is in the file `lab1/Collatz.java` which you updated in Lab 1 to print out the Collatz Sequence starting at `n = 5`. Your solution as it stood before you pulled the `lab4` starter code correctly printed out the Collatz sequence. The `skeleton` repository was updated to include the following buggy implementation of the `nextNumber(int n)` method, which returns the next number in the Collatz sequence after `n`:

```
/** Buggy implementation of nextNumber! */
public static int nextNumber(int n) {
    if (n  == 128) {
        return 1;
        } else if (n == 5) {
            return 3 * n + 1;
        } else {
            return n * 2;
        }
    }
```

Before we get ahead of ourselves, please make sure to ask your TA if you have any trouble at any point during this assignment; it's much better to get them involved earlier if you run into issues so that it's easier for them to recover. Your tasks are as follows:

Step 1

Resolve the merge conflict, *ensuring that the result of the merge conflict contains the **buggy** version of the method*. In other words, after you are finished resolving the merge conflict, `Collatz.java` should compile and contain the buggy implementation of `nextNumber`

Now, if you run `git log` (which lists the commits before the current commit, in order), you should see something like the following:

```
commit 8f0deeaef048f33a209f6f2fe5927a6fb04cc6cc
Merge: 225d73e 7aa1b6f
Author: Neil Kulkarni <neil.kulkarni@berkeley.edu>
Date:   Sun Feb 7 14:36:52 2021 -0700

    Merge branch 'master' of https://github.com/Berkel

    Fixed the merge conflict in lab4 to contain buggy

commit 7aa1b6fc79cb752e1ed844cd9cdd8c9c21e7f3d4 (HEAD
Author: Neil Kulkarni <neil.kulkarni@berkeley.edu>
Date:   Sun Feb 7 14:26:58 2021 -0700

    Added Lab 4 Starter Files

...

commit 4050fd80377d85aaea6c7cdb486e581d8c422534
Author: Neil Kulkarni <neil.kulkarni@berkeley.edu>
Date:   Sat Jan 30 22:56:58 2021 -0800

    Finished Lab 1! Collatz works!

...
```

Namely, you should see that the most recent commit is your merge commit that is the result of you fixing the merge conflict, and the second most recent commit is the commit where Neil added the Lab 4 Starter Files that you pulled from `skeleton`. You may have to go far back to see the commit you made when you finished Lab 1. For example, the commit that I made when I finished Lab 1 has the message "Finished Lab 1! Collatz works!". Your commit will very likely have a different commit message, and should have a different commit hash, author, and date. **Note down the commit hash of the commit in which you finished Lab 1**. Let's call this commit `lab1commit`, so we can refer back to it in future steps.

Step 2

Submit to the "Lab 4A: Git Exercise Part A" Autograder on Gradescope. It will verify that you correctly resolved the merge conflict to contain the buggy implementation of `nextNumber` in `Collatz.java`.

Step 3

Although your most recent commit includes a bug in `lab1/Collatz.java`, luckily at one point you committed a correct implementation of `Collatz`, namely in `lab1commit`! Since commits in git are snapshots of the state of files, `lab1commit` stores a snapshot of the correct version of `Collatz`. Don't believe me? Let's take a peek!

Your next step is to checkout to `lab1commit`. If you don't remember the syntax for this command, take a look at the materials linked above. Once you're there, type `git status`. You should see that you are in a detached HEAD state. If you don't remember what that is, please take a look at the materials above!

```
NeilKulkarni@Neils-MacBook-Pro sp21-s58 % git status
HEAD detached at 4050fd8
nothing to commit, working tree clean
```

Remember, in a detached HEAD state, you are free to look around in the current commit, but you shouldn't make any changes. Let's take a look at `lab1/Collatz.java`. You should see that it has your old solution! You can open this file in any manner you'd like, but I recommend the `cat` terminal command, which prints the contents of a file:

```
NeilKulkarni@Neils-MacBook-Pro sp21-s58 % cat lab1/Col
/** Class that prints the Collatz sequence starting fr
 *  @author Neil Kulkarni
 */
public class Collatz {
    public static int nextNumber(int n) {
        return n % 2 == 0 ? n/2 : 3*n + 1;
    }

    public static void main(String[] args) {
        int n = 5;
        System.out.print(n + " ");
        while (n != 1) {
            n = nextNumber(n);
            System.out.print(n + " ");
        }
    }
}
```

Step 4

Now, get out of the detached HEAD state by checking out to the most recent commit. You should verify that since we are back at the most recent commit that `lab1/Collatz.java` has the buggy implementation again:

```
NeilKulkarni@Neils-MacBook-Pro sp21-s58 % cat lab1/Col
/** Class that prints the Collatz sequence starting fr
 *  @author YOUR NAME HERE
 */
public class Collatz {

    /** Buggy implementation of nextNumber! */
    public static int nextNumber(int n) {
        if (n  == 128) {
            return 1;
        } else if (n == 5) {
            return 3 * n + 1;
        } else {
            return n * 2;
        }
    }

    public static void main(String[] args) {
        int n = 5;
        System.out.print(n + " ");
        while (n != 1) {
            n = nextNumber(n);
            System.out.print(n + " ");
        }
        System.out.println();
    }
}
```

Step 5

We've verified that `lab1commit` has the correct contents of `Collatz.java`, and we're back at the most recent commit which contains the buggy implementation of `Collatz.java`. Now, let's use `git checkout` to get `lab1/Collatz.java` to its state in `lab1commit`. If you've forgotten how to check out files, please review the materials at the beginning of lab!

When you check out a file, git automatically adds that file. Immediately after you've checked out, `git status` should return something similar to this:

```
NeilKulkarni@Neils-MacBook-Pro sp21-s58 % git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   lab1/Collatz.java
```

Go ahead and `cat lab1/Collatz.java` one more time. You should see that the solution is now updated! Go ahead and commit and push your changes.

Step 6

Submit to the Lab 4B: Git Exercise Part B autograder to get credit for doing this assignment.

Some parting words: what we did was very powerful. We took advantage of the fact that git stores snapshots of our files and used it to revert back to an older version! And, in the process, we learned how to resolve a merge conflict!

This information is not just theoretical. You can find it comes in practical use when you are working on your projects. As Prof. Hug has said often, it's okay (and encouraged!) to start over from scratch on your projects. Now, we have the means to do that! You can use the `git checkout` command to check out your project files to an older commit, before you made any changes to them. Specifically, if you want to checkout a project file from scratch (e.g. the `Model.java` file from Project 0), you can run the following command:

```
git checkout skeleton/master -- proj0/game2048/Model.j
```

Feel free to try this command out! **In general, `git checkout skeleton/master -- <file>` reverts a file back to its skeleton-code state**. Remember, you have commits of your progress on `proj0`, so you can always revert `Model.java` back to any of the states it was committed at (including the state at which the project was finished)! If you want to revert `Model.java` back to its most recent (and finished) state, you can run the following command:

```
git checkout master -- proj0/game2048/Model.java
```

In general, `git checkout master -- <file>` reverts a file back to its state at the most recent commit. Now you know how to restart a project, or revert a restarted project back to its state at `master`, if you were unhappy with the restart!

A Debugging Mystery

Another important skill to learn is how to exhaustively debug. When done properly, debugging should allow you to rapidly narrow down where a bug might be located, even when you are debugging code you don't fully understand. Consider the following scenario:

Your company, Flik Enterprises, has released a fine software library called `Flik.java` that is able to determine whether two Integers are the same or not.

You receive an email from someone named "Horrible Steve" who describes a problem they're having with your library:

```
"Dear Flik Enterprises,

Your library is very bad. See the attached code. It sh
but actually it's printing out 128.

(attachment: HorribleSteve.java)"
```

Using any combination of the following techniques, figure out whether the bug is in Horrible Steve's code or in Flik enterprise's library:

- Writing JUnit tests for the Flik library. If you want to try this, you will to create a new file in the `flik` directory and import `junit`. Look to the tests from the previous problems for how to do this.
- **Using the IntelliJ debugger, especially conditional breakpoints or breaking on exceptions, which you learned how to do in Lab 3!**
- Using print statements.
- Refactoring Horrible Steve's code. Refactoring means changing the syntax without changing the functionality. This may be hard to do since HS's code uses lots of weird stuff.

Once you find the bug, fix it and submit your code to the Lab 4: Debugging autograder. The autograder for this part uses hidden tests, so you will not be able to get any information about the bug from the AG. If you think you've fixed the bug, but are not passing the AG, please consult a TA.

Tip: JUnit provides methods `assertTrue(boolean)` and `assertTrue(String, boolean)` that you might find helpful.

Try to come up with a short explanation of the bug! Google is your friend since we have not covered this exact issue in Lecture. Discuss with your lab partner and check in with your TA or an AI to see if your answer is right (not for a grade).

Submission

For this lab assignment, there are three separate autograders. Lab 4A: Git Exercise Part A and Lab 4B: Git Exercise Part B confirm that you have completed the git exercise, and Lab 4: Debugging confirms that you found and fixed the bug from "A Debugging Mystery". This AG also checks style, so make sure your code passes the style check for each file! Remember, you can check style by right clicking a file > Check Style. For more information about when to submit to each AG, please see the corresponding parts of the spec.

Extension Requests: There are three autograders (Lab 4, Lab 4A, and Lab 4B). On Beacon, if you make an extension request for Lab 4, it will apply to all three autograders.

Full Recap

In this lab, we went over:

- Git Basics
- Merge Conflicts
- Detached HEAD States
- Checking Out Code with Git

- Debugging Using JUnit