Assignment 1

Commands in git bash

A commit in a git repository records a snapshot of all the files in your directory. Git also maintains a history of which commits were made when. That's why most commits have ancestor commits above them -- we designate this with arrows in our visualization.

Git branches

Branches in Git are incredibly lightweight as well. They are simply pointers to a specific commit -- nothing more. When we start mixing branches and commits, we will see how these two features combine.

Branches and merging

Merging in Git creates a special commit that has two unique parents. A commit with two parents essentially means "I want to include all the work from this parent over here and this one over here, *and* the set of all their parents."

Git rebase

The second way of combining work between branches is *rebasing.* Rebasing essentially takes a set of commits, "copies" them, and plops them down somewhere else. the advantage of rebasing is that it can be used to make a nice linear sequence of commits.

Moving around in git

Before we get to some of the more advanced features of Git, it's important to understand different ways to move through the commit tree that represents your project.
Head: HEAD is the symbolic name for the currently checked out commit -- it's essentially what commit you're working on top of.

Relative refs

Moving around in Git by specifying commit hashes can get a bit tedious. In the real world you won't have a nice commit tree visualization next to your terminal, so you'll have to use git log to see hashes.With relative refs, you can start somewhere memorable (like the branch bugFix or

HEAD) and work from there.Relative commits are powerful, but we will introduce two simple ones here
Moving upwards one commit at a time with ^
Moving upwards a number of times with ~<num>

Reversing changes in git

There are many ways to reverse changes in Git. And just like committing, reversing changes in Git has both a low-level component (staging individual files or chunks) and a high-level component (how the changes are actually reversed). Our application will focus on the latter.
There are two primary ways to undo changes in Git -- one is using git reset and the other is using git revert.

Moving work around

Git cherry pick: The first command in this series is called git cherry-pick

Locally stacked commits

All of these debugging / print statements are in their own commits. Finally I track down the bug, fix it, and rejoice!
Only problem is that I now need to get my bugFix back into the master branch. If I simply fast-forwarded master, then master would get all my debug statements which is undesirable. All of these debugging / print statements are in their own commits. Finally I track down the bug, fix it, and rejoice!

Git Tags

Git tags support this exact use case -- they (somewhat) permanently mark certain commits as "milestones" that you can then reference like a branch.

More importantly though, they never move as more commits are created. You can't "check out" a tag and then complete work on that tag -- tags exist as anchors in the commit tree that designate certain spots.

Specifying parents

Like the ~ modifier, the ^ modifier also accepts an optional number after it.
Rather than specifying the number of generations to go back (what ~ takes), the modifier on ^ specifies which parent reference to follow from a merge commit. Remember that merge commits have multiple parents, so the path to choose is ambiguous.

Git will normally follow the "first" parent upwards from a merge commit, but specifying a number with ^ changes this default behavior.