Section 1: Basics

Exercise 1: Adding a file to new repository

- In github, find the menu item that creates a new repository.
- Create a repository, we will refer to it as “new_repo”. Add a readme so it can be immediately checked out. Now clone it to your machine.

```
git clone <repo url>
```

- You can also create a repository locally. If you don’t have a github account, or if there is a problem reaching github, you can do this for the exercises.

```
mkdir new_repo
cd new_repo
git init
```

- Create a file under new_repo, and tell git to commit the file. Committing file modifications uses the same git commands.

```
echo ‘testfile’> file.txt

git add file.txt
OR
git add . -A  (adds all changes)

git commit -m “Added file.txt”
```

- If you created your repository in github, push your changes to it. When you next clone the repo, it’s history and contents will match what you have now.

```
git push origin master
```

Exercise 2: Crafting commits

- Our goal here is to build small, easy to understand commits. You probably don’t want to commit code until your tests pass, but you also don’t want to commit one big blob at the end.

- Add a new file and modify the old one. Notice that the status check shows the two changes differently. Use the -u switch to only commit the modification.

```
echo ‘newfile’ > file2.txt
echo ‘I am an update’ >> file.txt
git status -s
git add -u
git commit -m “Updated file.txt”
```

- Like modifications, file deletions are picked up with -u by default. This is convenient because non-existent files don’t auto-complete in the terminal.
- Lets do something with the second file to get it out of the way. Assume you forgot to commit file2.txt the first time, in this case you can quickly fix this with the following:

```
git add file2.txt
git commit --amend
```

- Now, suppose another case. You’ve made a mess and want to start all over. To do this you must unstage changes, remove changes from disk, then delete untracked files. Add some new files and/or change your current ones, then do the following:

```
git reset HEAD .
git checkout .
git clean -f
```

Exercise 3: Branching basics

- Create a new branch, list the current branches, then push our branch to github.

```
git status  (should be on branch master)
git checkout -b new_branch
git branch  (should be on branch new_branch)
git push origin new_branch
```

- Getting changes from master into your branch:

```
echo ‘Another update’>> file.txt
git checkout master
git add .
git commit -m “Updating file.txt for merging”

git checkout new_branch
git merge master
```

- If you do ‘git log’, it should have both a merge message and “Updating file.txt for merging” at the top, which was brought it from master.
- You can merge from other branches too, not just master.

Section 2: Using git history

Exercise 1: Resetting to a given commit

- Resetting moves the HEAD pointer back to a previous commit.
- Commit an arbitrary change to your repository. Using ‘git log’, you can copy and paste the sha1 hash that corresponds to a commit. Example: b1682119643c1d5de38c4bf02fc54ccf3811ecc8

- Copy the commit right beneath your latest one.

```
git reset <commit hash>
```

- This takes the history back from before your last commit, but leaves your change intact. git log will now show the commit you copied at the top. Your changes are still on disk by default, so you can commit again without re-writing code. Although the latest commit is not visible in git log, it still exists in git’s datastore.

```
git reflog
```

- This shows which commits HEAD has pointed to. Like before, you can copy the first few digits of a hash, and git reset back to it.

- Hard resets do the same thing, but in addition to history, they reset your filesystem. If you want your repo to look exactly as you did at a given commit, use a hard reset.

```
git reset --hard <commit hash>
```

- Like with soft resets, you can look at the reflog and undo this.

Exercise 2: Applying individual commits

- Like before, lets look at the top of git log. Copy the commit hash at the top.

```
git revert <commit hash>
```

- This generates a diff that undoes the changes in the commit you specified. It then commits that diff. You can achieve the same effect with git reset --hard to go back a commit, but with revert you leave the unwanted change in git log and is more clear what you did. Reverts themselves can also be reverted, but I recommend against this. Commits with the word ‘Revert’ four times at the front of the message get old.

- Another way to undo a revert is with git cherry-pick. Using the same commit hash you used for the revert:

  ```
  git cherry-pick <commit hash>
  ```

- There need not be a revert in order to use cherry-pick to apply a commit. To demonstrate this, try committing a 2-line file to your repository, and set aside the hash for that commit. Delete one line from this file, and commit your change. Delete the remaining line from your file, which should now be empty, and commit. You can restore the file completely with:

  ```
  git cherry-pick <commit hash>
  ```

- You can imagine that you made a commit with two helper methods, which got deleted at different times, and you want to restore them. You can do this with one cherry-pick or two reverts, whichever is easier at the time.