CSE 101
Introduction to Data Structures and Algorithms
GitLab Tutorial

All programming assignments in this class will be submitted through the UCSC GitLab server

https://git.ucsc.edu

Go to ITS GitLab, follow the login link, and create an account. Your username should automatically be set to be your **cruzid**, which is the string before the **@** in your **@ucsc.edu** email address. Occasionally a username will differ slightly from this form, having the integer 1 after the username, i.e. **cruzid1**. If this happens, open a request ticket at https://slughub.ucsc.edu/its to make your username identical to your **cruzid**. Creating a properly named account on **git.ucsc.edu** is step zero in this tutorial.

While Git is, strictly speaking, not part of the content of this course, it is an essential professional skill for software developers. However, if you want to completely bypass learning Git commands, you can submit your work directly through the **git.ucsc.edu** web interface and skip the rest of this tutorial.

Git is a *Distributed Version Control System*, which means that multiple copies of a project can be maintained in different locations, then integrated and synchronized. At the enterprise level, this might entail the work of hundreds of programmers, all seeking to manage and reconcile their different versions of the project. In this class there is just one programmer, namely you. Your *project* consists of all of the assignments you submit for grading in this class. I recommend that you maintain only two copies of your project. Your *local repository* is on your personal computing device, which you directly control and maintain, and on which you develop your programs. Your *remote repository* is on the server **git.ucsc.edu**, which is maintained by Information Technology Services (ITS). You will synchronize these two repositories by runing Linux commands begining with the keyword `git`.

If you are completely new to Linux, or to the concept of a command line interpreter, you have some catching up to do. I suggest you read one of the Linux tutorials on the class webpage. The remainder of this tutorial assumes that your local repository will be created in a Linux environment that you control. This could be a Linux virtual machine running on your laptop, or Windows Subsystem for Linux (WSL) running on your Windows 10 or Windows 11 PC, or a Linux cloud server on which you have an account (see here for a list of such services). You may also maintain the local repo within your personal Windows or Mac file system. The instructions are largely the same as those presented here, except that you would need to first install git on your machine. I recommend that you *not* maintain multiple local repositories, one on a Linux server say, and one on your device, though it is possible. Attempt this only if you are experienced with git and are sure you know how.

Set up ssh keys

The next step is to set up a pair of public-private ssh keys to facilitate secure communication between local and remote repositories. Log on to your Linux system and type the following command.

```
$ ssh-keygen -t rsa -b 2048 -C "label"
```

The dollar sign `$` represents the command prompt, and you do not type it. Also `"label"` is a comment that you choose. This comment can be anything you like, but typically it is used to identify the computing device you are on. If you are setting up a repository on your own computer, you might choose the name of your local machine as the label. The result of the above command will be

```
Generating public/private rsa key pair.
Enter file in which to save the key (/..../userid/.ssh/id_rsa):
```

or something similar (`userid` will be your username on your Linux system). Press return to select the default location `userid/.ssh/id_rsa`, (the file `id_rsa`, within the directory `.ssh`, within your home directory `userid`.) Navigate to the directory `.ssh` (do `cd` then `cd .ssh`) and type

```
$ cat id_rsa.pub
```

to see the public half of the pair. The private half is in the file `id_rsa`. Do not share the private key with any person or computer, and do not alter either file.

Now login to **git.ucsc.edu** and open the web portal. At the top of the navigation panel on the left, you will see a picture or icon representing your account. Clicking on it opens a menu from which you should select **Preferences**. Go to the left-hand pane on the settings page and select **SSH Keys**. Press `Add new key`, then copy-paste the public key in `id_rsa.pub` into the `key` box. Give it a `title` (and an optional expiration date if you like), then press `add key`. You have now established a secure method of communication between your Linux system and your account on **git.ucsc.edu**.

Set up your local repository
Return to your home directory on Linux (by typing `cd`), then type

```
$ git config --global user.name "<your first name> <your last name>"
$ git config --global user.email "cruzid@ucsc.edu"
```

where `cruzid@ucsc.edu` is your UCSC email address. Now create a subdirectory within your home directory called `cse101`, where you will keep all of your work for this class, then cd into it.

```
$ cd            # return to your home directory
$ mkdir cse101  # make a subdirectory called cse101
$ cd cse101     # change your working directory to cse101
```

Note that everything after `#` on a line is a comment and need not be typed. Now do

```
$ git clone git@git.ucsc.edu:courses-instruction/cse-101/spring26/cruzid.git
```

where as usual, `cruzid` is your CruzID. If you get an error message resembling

```
Cloning into 'cruzid'...
remote:
remote: ========================================================================
remote:
remote: The project you were looking for could not be found.
```

```
remote:
remote: ========================================================================
remote:
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

it means that I have not yet created your repository on **git.ucsc.edu**. I will be running a script periodically (every few days) that creates student repositories, but I can't create a repository for you if you don't have an account. This is why I run it repeatedly. If you have this error, you should pause and try again the next day.

Assuming you did not get an error, the above command should produce the following output.

```
Cloning into 'cruzid'...
remote: Enumerating objects: 10, done.
remote: Counting objects: 100% (10/10), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 10 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (10/10), done.
```

Once this is done, a new subdirectory within `cse101` called `cruzid` has been created.

```
$ ls     # list the contents of the current directory
cruzid
```

This subdirectory is your local repository. Make `cruzid` your current directory, and just in case you have become confused as to your location, type `pwd` to see the full path name of the repository.

```
$ cd cruzid  # change directory to cruzid
$ pwd        # print working directory
/..path to Linux home directory../userid/cse101/cruzid
```

root of the file system — your home directory — local repository

Note the distinction between `userid`, which is your Linux username, and `cruzid` which is your account name on **git.ucsc.edu**. Currently, this directory contains several files placed there when your repository was created. To see the full contents do

```
$ ls -a  # same as ls, but include files that begin with dot .
```

You can ignore the contents of this directory at the moment. Create a subdirectory for programming assignment 1, called `pa1` and `cd` into it.

```
$ mkdir pa1
$ cd pa1
```

The new directory `pa1` is within the directory `cruzid`, but is not yet a part of the repository. However, we cannot add an empty directory to a repository, so we must first insert some files. Eventually you will place all of your files for `pa1` here, but for now a couple of empty files will suffice.

```
$ touch file1
$ touch file2
```

The Linux command `touch blah` creates a new empty file called `blah`, if `blah` does not exist. View the manual pages for `touch` (do `man touch`) to see what it does if file `blah` already exists. Now we have a directory within the pa1 directory containing two empty files. To add it to the repository, do

```
$ git add .
$ git commit -m "initial commit on pa1"
```

The dot `.` in this context means your current working directory, which you'll recall is `pa1`. The command `git add .` places the directory `pa1`, with its contents `file1` and `file2`, into a staging area called the *index*. The command `git commit` alters the local repository so that it now includes the new items. The option `-m "message"` attaches a comment to this commit, so users will be able to follow the history of changes. At this point, all that has been changed is the local repository. To synchronize with the remote repository, do

```
$ git push
```

This command performs the initial synchronization between your local repo and the remote repo on **git.ucsc.edu**. Subsequent synchronizations can be done with the same command.

Observe that `add` has no terminal output, but `commit` and `push` do, which was not included above. To verify that the remote repository has changed, go the web portal on **git.ucsc.edu** again, go to **projects** in the left hand navigation pane, then navigate to **courses-instruction/cse-101/spring26/cruzid** to see the contents of your remote repository.

This combination of git commands: `add`, `commit` and `push` will be your main tools for doing subsequent submissions to `pa1`. For instance, to add a third file, do

```
$ touch file3
$ git add file3      # git add . would be fine here too
$ git commit -m "add file3"
$ git push
```

and observe the changes to the web portal. Now try deleting a file using `git rm`, then `commit` and `push`, as before. Intersperse the command `git status` between the others to see a readout of changes to the repository.

```
$ git rm file1   # remove file1
$ git status
$ git commit -m "delete file1"
$ git status
$ git push
$ git status
```

Let's add some content to one of the remaining files, and use `git diff` to display the difference. A simple way to append text to a file is `echo` with output redirect >.

```
$ echo "some content for file2" > file2
$ cat file2
some content for file2
$ git diff
diff --git a/pa1/file2 b/pa1/file2
index e69de29..877005d 100644
--- a/pa1/file2
+++ b/pa1/file2
@@ -0,0 +1 @@
+some content for file2
```

The output of `git diff` is a little bit cryptic, but this article may help explain it somewhat. Ordinarily we won't run `git diff`, but it's good to remember that all git does, at the most base level, is to track changes to files and directories. The same combination of `add`, `commit` and `push` will synchronize the local repository with the remote.

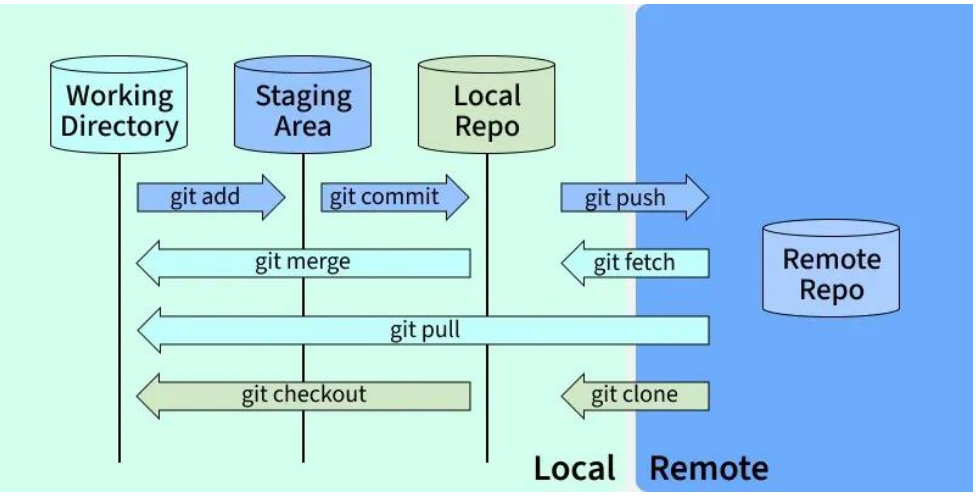
```
$ git add .
$ git commit -m "content for file2"
$ git push
```

At this point, you can add your source files for `pa1` (and also remove `file1` and `file2`), then perform the now familiar `add`, `commit`, `push` combination.

When it is time to start working on pa2, do

```
$ mkdir pa2
$ cd pa2
```

then use the `add`, `commit` and `push` commands to place new work in the local repo, and synchronize with the remote. Repeat the whole process for `pa3`, … , `pa8`. After you've familiarized yourself with some additional commands, the following is a good mental image to keep in mind.



Attend the TA/tutor/instructor office hours to get help with this tutorial if anything is unclear. Git has a rich command set with a fine level of control over your projects, which we've only glimpsed. The Git Book is one place to continue learning. Another resource on Linux is the git tutorial.

```
$ man -7 gittutorial
```

There are many others to be found using Google.