Homework 6
Git II

Due: Wednesday, October 24th, 11:59PM (Hard Deadline)

Submission Instructions

For this assignment, submit parts 1 and 2 to GitLab and part 3 to Gradescope.

Optional Reading

Git is a purely functional data structure, by Philip Nilsson, from Jayway.

`http://www.jayway.com/2013/03/03/git-is-a-purely-functional-data-structure/`

I highly encourage reading this when you have some time to read the article carefully and think deeply about the material. This article presents an excellent way of thinking about git and how it operates.

1 Evaluating git usage

Earlier this semester, we asked you to use git with at least one project. Now you will set up that project to be shared with the course staff. Visit `https://gitlab.eecs.umich.edu` and create a new project named **exactly** `c4cs-f18-wk6`. Be sure to create this project as **Private** (not Interal or Public).

Add this new repository as a remote (`git remote add ...`) to your existing project.

Push the project to this new remote.

In GitLab, grant **Reporter** permission to `tarunsk`, `amrith`, `cyanliu`, `sltries` and `mmdarden`. (Choose "Members" from the drop-down list from the settings gear in the top samkhanright to manage this permission).

We will run a test script (you can see the grading script here), checking our access to everyone's repository on Thursday.

Some key factors we are looking for:

1. Commit length. Not too long for the title of the commit message, and not too short. Also including more description in the body of the commit message is looked for.
2. Number of commits. Only having one or two commits in a repository doesn't mean you really used git effectively. In Homework 2 we asked you to begin using git for some sort of a project. This project should have at least five commits in it for it to be considered for full credit.

Again, for more insight, check out the script used to grade this homework assignment.

2 Handling merge conflicts

2.1 Content Conflict

Clone https://gitlab.eecs.umich.edu/c4cs/c4cs-git-conflict1.git

This repository has a `master` branch and a `merge_me` branch that have diverged. Merge the `merge_me` branch into `master`, resolving the conflict.

When you are done, running `bash test.sh` should print “Success” and running `python main.py` should print something reasonable (if factually inaccurate now, hooray!).

Create a new repository in your GitLab named **exactly** `c4cs-f18-conflict1`. Be sure to create this project as **Private** (not Internal or Public).

Push your changes to your new repository. (*Note: This will be a different “remote”*)

In GitLab, grant **Reporter** permission to `tarunsk`, `amrith`, `cyanliu`, `sltries` and `mmdarden`. (Choose “Members” from the drop-down list from the settings gear in the top right to manage this permission).

We will run a test script, checking our access to everyone’s repository on Thursday night.

First, clone the repository:

```
$ git clone https://gitlab.eecs.umich.edu/c4cs/c4cs-git-conflict1.git
```

This brings down the `master` branch onto our machine and adds `c4cs/c4cs-git-conflict1` as a remote. We can verify this:

```
$ git remote -v
origin  https://gitlab.eecs.umich.edu/c4cs/c4cs-git-conflict1.git (fetch)
origin  https://gitlab.eecs.umich.edu/c4cs/c4cs-git-conflict1.git (push)
$ git branch
* master
```

Now we want to merge in the branch `merge_me`. However, we don’t have this particular branch locally, as is seen from the `git branch` command above. It resides on the remote server `origin`.

```
$ git merge origin/merge_me
Auto-merging test.sh
CONFLICT (content): Merge conflict in test.sh
Auto-merging main.py
CONFLICT (content): Merge conflict in main.py
Automatic merge failed; fix conflicts and then commit the result.
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   main.py
        both modified:   test.sh

no changes added to commit (use "git add" and/or "git commit -a")
```

Let's have a look at the two files with conflicts. Notice the markers ««« HEAD ==== »»» origin/merge_me. These are conflict markers. Everything below HEAD is what was in HEAD (which points to the master branch in our case) before the merge. Everything below the equals and above origin/merge_me was is what was in that branch.

Important note: The conflict markers are automatically inserted by git but should **never** be committed.

Git tries its best to merge intelligently, but sometimes if the same areas of a file change it needs a human to reconcile the differences. This is an example of one such case. The correct solutions for each file are below:

```
$ cat main.py | nl
...
37  if __name__ == '__main__':
38          print_welcome()
39          print('')
40          print('According to current estimates, the diag construction will be done:')
41          time = compute_time(2) # 2 is March
42          print(pretty_print(2, 2016, time))
$ cat test.sh | nl
 1  diff <(python main.py) golden.txt
...
```

In main.py we wanted to keep the first part of our message from master and the second better estimate from merge_me. In test.sh, the conflict was resolved by keeping everything from master. You can verify this by running ./test.sh and seeing the success message.

To complete the merge:

```
$ ./test.sh
Success
$ git add test.sh main.py
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

        modified:   main.py
$ git commit
[master 810c3af] Merge remote-tracking branch 'origin/merge_me'
```

You'll notice that test.sh does not appear in the git status. This is because it's actually identical to the master branch's copy. i.e. nothing changed, so there are no changes to be committed for that file.

2.2 File Path Conflict

Repeat the same steps for https://gitlab.eecs.umich.edu/c4cs/c4cs-git-conflict2.git

Be sure to read over the commit history so that you are sure that the result of your merge has the right data!

We start out very similar to the previous question.

```
$ git merge origin/merge_me
CONFLICT (modify/delete): sales-march-1st deleted in origin/merge_me and
modified in HEAD. Version HEAD of sales-march-1st left in tree.
Automatic merge failed; fix conflicts and then commit the result.
$ git status
```

```
On branch master
Your branch is up-to-date with 'origin/master'.
You have unmerged paths.
  (fix conflicts and run "git commit")

Changes to be committed:

        new file:   sales-2016-03-01
        new file:   sales-2016-03-02
        new file:   sales-2016-03-03

Unmerged paths:
  (use "git add/rm <file>..." as appropriate to mark resolution)

        deleted by them: sales-march-1st
```

It appears that `sales-march-1st` doesn't fit the pattern of our other files. Intuition tells us it should probably be deleted, but we should double check.

```
# command below shows logs from merge_me & describes files changed in each commit
$ git log --name-status origin/merge_me
commit 0e9f1fba2bd00827dad94bc44585ed8b9f2f2c26
Author: Pat Pannuto <pat.pannuto@gmail.com>
Date:   Fri Mar 11 10:49:17 2016 -0500

    Add March 3rd. Change naming scheme

    Using human-friendly names "march 1st" makes it harder than needed
    for computers to parse the dates. It will also break the simple
    sorting of tools like `ls` once you get to two-digit dates.

    Change the naming scheme to parse-friendly, sort-friendly numbers
    instead.

A       sales-2016-03-01
A       sales-2016-03-02
A       sales-2016-03-03
D       sales-march-1st
D       sales-march-2nd

commit f8e2f6933c9ea0f20bdc361934ebed1c55c53c40
...
```

```
commit 302584e32cc72978c6febb1b138d2afd892a7a25
Author: Pat Pannuto <pat.pannuto@gmail.com>
Date:   Fri Mar 11 10:52:45 2016 -0500

    Update March 1st with real sales data

    Replace the placeholder data with the real sales data from March 1st.

M       sales-march-1st

commit 1710c112b061bb1d4c43d0366b73a225fe93eb91
Author: Pat Pannuto <pat.pannuto@gmail.com>
Date:   Fri Mar 11 10:14:18 2016 -0500

    Add initial sales calculation (note: buggy!)

    This is the point in history where the two branches will diverge.

M       main.py
A       sales-march-1st

commit b469c96bae4424dfdc33c7aae253a663bd56735e
Author: Pat Pannuto <pat.pannuto@gmail.com>
Date:   Fri Mar 11 10:08:56 2016 -0500

    Initial commit.

A       README.md
A       main.py
```

Interpreting commit messages modified files tells us quite a bit. In `fe93eb91`, we committed buggy data to the (now deleted) sales file. We then see that in `892a7a25` we update our file with real data, so `master` definitely contains old data. Finally, in `f2f2c26` we see why the files were renamed. We can confidently delete `sales-march-1st`.

```
$ git rm sales-march-1st
sales-march-1st: needs merge
rm 'sales-march-1st'
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

        new file:   sales-2016-03-01
        new file:   sales-2016-03-02
        new file:   sales-2016-03-03
        deleted:    sales-march-1st
$ git commit
[master f58156c] Merge remote-tracking branch 'origin/merge_me'
```

3 Markdown

Markdown is a widely used lightweight markup language designed to be easily converted into many common formats (eg. HTML or pdf). GitHub uses its own specification called GitHub Flavoured Markdown (GFM) for everything from README docs to pull request comments, so it's important to know some basic Markdown for if you make changes in a public repository.

In the spaces given below, write the Github Flavoured Markdown that would produce the following:

1. **A level 1 header with the text "HW 6 Markdown"**

   ```
   # HW 6 Markdown
   ```

2. **An inline link to** `https://c4cs.github.io` **with the text "C4CS site"**

   ```
   [C4CS site](ttps://c4cs.github.io})
   ```

3. **An unordered list with 2 items, one in** *italics* **and one bold**

   ```
   * *Item 1 (italics)*
   * **Item 2 (bold)**
   ```

4. **A code block with c++ syntax highlighting and this code snippet:** `int rand() { return 3; }`

   ~~~
   ```cpp
   int rand() {
   return 3;
   }
   ```
   ~~~

5. **A 2x2 table with the following specifications:**
   - The first row should be a header row
   - The left column should be left aligned
   - The right column should be right aligned

   ```
   | Left-aligned | Right-aligned  |
   | :---         |           ---: |
   | Content1     |     Content2   |`
   ```