Git Competency

The purpose of this assignment (more of a walkthrough really) is to learn and practice a few common git skills that will help you navigate source control in a team.

Enter your answers for **bold questions** directly in this document. One submission per group. Complete all other tasks on your gitlab repo.

**Setup**

1. All Users:
   1. Create student GitHub accounts here: https://education.github.com/
2. User 1:
   1. Create a private GitHub repository called GitAssignment and add the other 3 group members with read/write access to your repo. This is the equivalent of Tortoise Git > Create repo here, and will be the shared repo that all group members use.
   2. Add the instructor to your repo as well: my user name is mattwarnerneumont
   3. **What is the URL to your github repo?** **https://github.com/Error503/GitAssignment** (it will look something like https://github.com/yourusername/GitAssignment)

**Part 1 Non-Conflicting Changes**

1. Each user in the group completes all the steps in this section except as noted.
2. Clone the shared GitHub repo to your local machine to a folder called GitAssignment. Do all your work for this assignment inside that folder.
3. Add a file to that folder called yourlastnamefirstinitial.txt. Mine would be warnerm.txt.
4. Add two separate lines to that file saying anything you wish (PG please!)
5. Commit and push that new file.
   1. If you get tired of typing your username and password, you can save them. Tortoise Git > Settings > Git > Credential > Credential Helper: Wincred – this repository only. The next time you enter your user name and password they’ll be saved for that repository.
6. If at any time git says you can’t push because of remote changes, git > pull, then push. This brings your local working directory up to date with the shared repo. If you read the error message it actually tells you this:

hint: Updates were rejected because the remote contains work that you do

hint: not have locally. This is usually caused by another repository pushing

hint: to the same ref. You may want to first integrate the remote changes

hint: (e.g., 'git pull ...') before pushing again.

...

git did not exit cleanly...

1. Have one user add a file called Team.txt with the names of all team members sorted alphabetically and named User N. Commit and push.

User 1: Alfred Albott

User 2: Beta Carotene

User 3: etc...

1. All users Pull.
2. Make a one-word change to yourlastnamefirstinitial.txt.
3. Tortoise Git > check for modifications > double click the yourlastnamefirstinitial.txt file.
   1. The diff viewer should open, showing the changes you made and are about to commit.
      1. I usually check \_every\_ file I’m about to commit in this way to make sure I really mean to commit all the changes.
4. Commit and push.
5. After everyone has successfully completed step 13, everyone pull.
   1. Tortoise git > Git show log
   2. **Add a screen shot of the log below.**
   3. **Describe what the revision tree looks like and why.**

**Part 2: Conflicting Changes**

1. Make sure everyone is up to date. (No pending local changes, pull and push)
2. Check out [these walkthroughs](https://www.youtube.com/playlist?list=PLeWlGN6nzyn6mpkFzaWwrGXvLua2phJFF), where I demonstrate how two users can edit the same file and resolve their conflicts.
3. Have user 1 add lines numbered 10 through 19 to the end of Team.txt. Save but **don’t** commit.

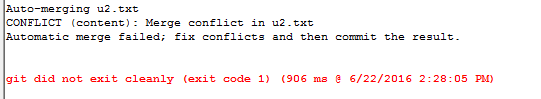
Line 1

Line 2

Line 3

Etc...

1. User 2 add lines numbered 20 through 29 to the end of Team.txt. Commit and push.
2. User 3 add lines numbered 30 through 39 to the end of Team.txt. Save but **don’t** commit.
3. User 4 add lines numbered 40 through 49 to the end of Team.txt. Save but **don’t** commit.
4. **Users 1, 3, 4: Try to pull. What happens? Why does git require you to commit your changes before pulling?**
5. Users 1, 3, 4: Commit your changes to Team.txt, then pull. You should get a conflict:



1. A conflict occurs when users make changes to the same part of a file. Git is unable to automatically resolve these conflicts. Instead the problem file is marked as conflicted. Tortoise Git > Check for modifications to see the conflicted files. Double click on a file to open the GitMerge toolUse the diff viewer to view the conflicting changes.
2. Resolve the conflicts so that all of lines 1 through 49 are retained in the final master version of Team.txt. You’ll see the “First in wins” principle working here as each user tries to race to get their 10 lines committed and pushed before the next user. This will work easiest if you don’t all try to commit and push your changes at the same time. The videos linked above walk through how to resolve conflicts.

**Part 3. Aieeeee! Undoing local changes and bad commits.**

1. Everyone:
2. Make a change to one line in Team.txt in your line number block (user 2 will change something between lines 20 through 29). Save the file (don’t commit or push).
3. Tortoise Git > check for modifications. Double click Team.txt in the Tortoise Modifications window. View your changes using the diff viewer, then close the viewer. This is how you can see what has changed since your last commit.
4. Tortoise Git > check for modifications > right click on Team.txt and revert. This process mimics a change that you were thinking about but that went bad. As long as you haven’t committed yet, you can use revert to undo all your changes since your last commit, even if you can’t use IDE features to undo your changes.
5. Make another change to one line in your assigned block of lines in Team.txt. Save the file. This time commit. (don’t push)
6. Tortoise Git > show log. Right click on your most recent commit > revert change by this commit. This takes whatever happened in that commit and undoes it on your working directory.
7. Just like in step 3, you can view the reverted changes before you commit using > check for modifications. This is another reason why I commit broken code sometimes so I can clearly see what the fix was. (Aside: when committing broken code make that fact clear in your commit message)
8. Commit your change.
9. Pull then Push. You might have some conflicts to resolve depending on how you changed and undid changes to Team.txt.
10. Use show log to make sure everyone is up to date on the same revision.

**Part 4: Branching and merging.**

Branching allows developers to work independently, committing **and** pushing their code without affecting other developers.

1. Everyone: make sure you have no pending local changes and pull.
2. User 3: Tortoise git > switch/checkout > Create new branch > GitAssignmentFeature1
3. Add a new file called feature1.txt with some text inside. Commit and push.
4. User 4: Tortoise git > switch/checkout > Create new branch > GitAssignmentFeature2
5. Add a new file called feature2.txt with some text inside. Commit and push.
6. User 1: Pull. **Can you see any of the changes from users 3 and 4?**
7. User 1: Switch/Checkout to GitAssignmentFeature1.
   1. Add some code to feature1.txt, commit and push.
   2. Switch/Checkout to master. You’ve just helped a team member without messing up your local (for long) to whatever they were working on.
8. User 3: Pull. Look! User 1’s changes!
   1. This is the equivalent of working on your own feature, asking another developer for help, and receiving help without messing up the other developer’s world.
9. User 2: Stay on the Master branch (you should already be there). Make a change to Team.txt, adding text on line 26. Commit and push.
10. Everyone: Pull. **Can you see user 2’s changes?**
11. User 2’s change represents a main line change that everyone should get on their feature branches. When changes are happening on separate feature branches, it can be helpful to wait between merges (so you’re not interrupted by every change), but not for too long (so merging isn’t a huge pain with a bazillion conflicts).
12. User 3: Tortoise Git > Merge to get the changes from the Master branch onto the GitAssignmentFeature1 branch. Commit and push.
13. User 1: Switch/Checkout to GitAssignmentFeature1. Show log. You should be able to see the merged changes from the master branch locally on the feature branch.
14. Everyone: Switch/checkout to master, pull.
15. User 3: We’re keeping your feature. Merge on the master branch to pull your changes from GitAssignmentFeature1 to master. This is how features are usually added to the master line in git. We bring the feature branch up to speed with main (merge main to feature branch), then we merge the feature branch into main (feature branch to main).
16. User 4: Sorry! We’ve decided not to add your feature to master. Your branch will sit there forever. If you ever want it, switch/checkout to it. But it will sit, unused, sad and alone.
17. **Why bother branching and merging? Isn’t it easier for everyone to work on master?**

**Last, but not Least: The Last, but not Least Section.**

1. **Take a screen shot of your tree with \_all\_branches\_showing. Paste it here.**
2. **How was the walkthrough? What did you learn? How might I improve it next time?**