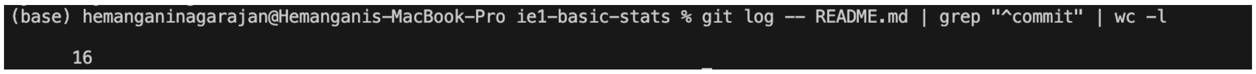
Team: **CodeCraftersH**
Members: **Harshita Loganathan, Hemangani Nagarajan**

https://github.com/HEMANGANI/ie1-basic-stats

Note: Our basic-stats-fork folder (specifically its .git folder) contains all of the following:
1. branch
2. cherry pick
3. rebase
4. merge
5. commit a change to the README file
6. reset
7. revert

1. Include the answers to the 6 queries from Section 1.

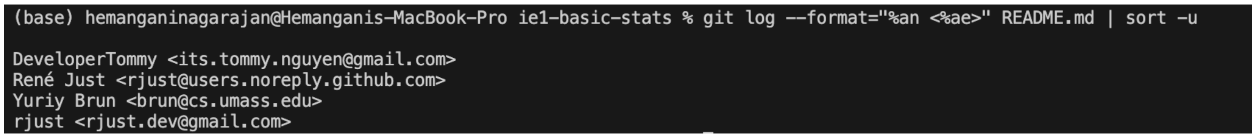
1) 16



```
(base) hemanganinagarajan@Hemanganis-MacBook-Pro ie1-basic-stats % git log -- README.md | grep "^commit" | wc -l

      16
```

2)

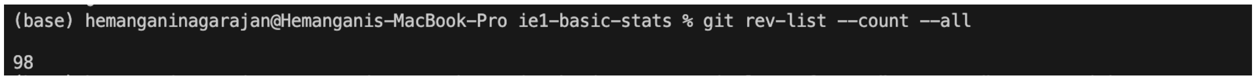


```
(base) hemanganinagarajan@Hemanganis-MacBook-Pro ie1-basic-stats % git log --format="%an <%ae>" README.md | sort -u

DeveloperTommy <its.tommy.nguyen@gmail.com>
René Just <rjust@users.noreply.github.com>
Yuriy Brun <brun@cs.umass.edu>
rjust <rjust.dev@gmail.com>
```

3) commit c2111cc0d37bfde779a317e533d3a5e68b8ed9e3

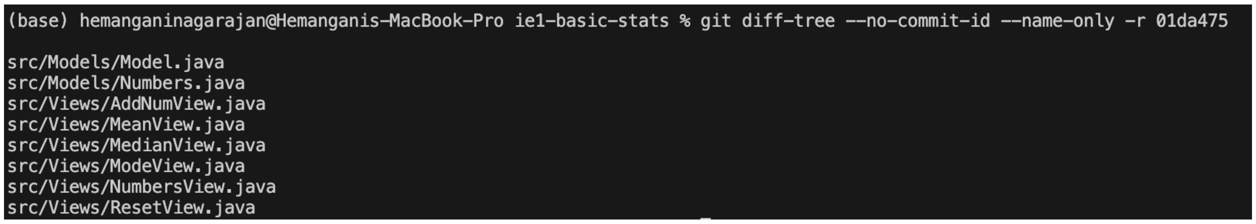
4) 98



```
(base) hemanganinagarajan@Hemanganis-MacBook-Pro ie1-basic-stats % git rev-list --count --all

98
```

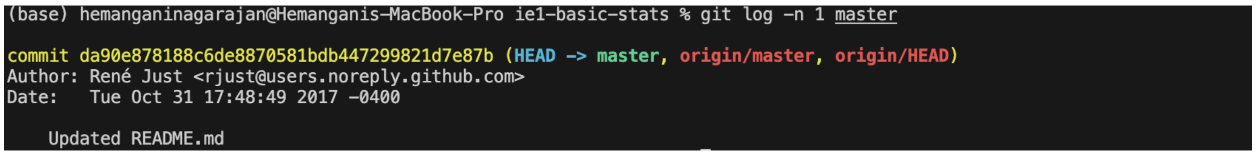
5)



```
(base) hemanganinagarajan@Hemanganis-MacBook-Pro ie1-basic-stats % git diff-tree --no-commit-id --name-only -r 01da475

src/Models/Model.java
src/Models/Numbers.java
src/Views/AddNumView.java
src/Views/MeanView.java
src/Views/MedianView.java
src/Views/ModeView.java
src/Views/NumbersView.java
src/Views/ResetView.java
```

6)



```
(base) hemanganinagarajan@Hemanganis-MacBook-Pro ie1-basic-stats % git log -n 1 master

commit da90e878188c6de8870581bdb447299821d7e87b (HEAD -> master, origin/master, origin/HEAD)
Author: René Just <rjust@users.noreply.github.com>
Date:   Tue Oct 31 17:48:49 2017 -0400

    Updated README.md
```

2. Detail the comprehensive procedure for adding a file named `MinMaxCalculation.java` to a remote Git repository. Your answer should cover the following key areas:

   - Commands executed on your local machine to initiate the process.
   - How to manage merges, particularly in the context of collaborative development.
   - Steps for pushing the changes to the remote repository.
   - Techniques for identifying the specific commit related to this operation.

   In your explanation, please integrate best practices such as appropriate naming conventions for branches and commits, as well as the use of .gitignore to exclude files that should not be versioned.

First, We navigated to the local copy of our ie1-basic-stats repository:

```
cd path/to/ie1-basic-stats
```

Next, We ensured I had the latest updates from the remote:

```
git fetch origin
git checkout main
git pull origin main
```

For clarity and to manage my work better, We created a new branch named feature-min-max-calculation:

```
git checkout -b add-minmax-calculation
```

Then, We added the MinMaxCalculation.java file to the repository. We ensured that our .gitignore was up-to-date, preventing unwanted files like logs or compiled classes from being versioned.

After writing and testing the code, I staged the file:

```
git add MinMaxCalculation.java
```

We committed the changes with a descriptive message:

git commit -m "Add functionality for calculating min and max values"
Before merging my changes back into the main branch, We wanted to ensure compatibility with any updates made by my teammates. So, We fetched the latest updates again:

```
git fetch origin
```

We then rebase my feature branch on top of the main branch:

```
git rebase origin/main
```

During the rebase, We ensured any merge conflicts were resolved. Afterward, We pushed my feature branch to the remote ie1-basic-stats repository:

```
git push origin add-minmax-calculation
```

Finally, We checked my commit history using:

```
git log --oneline
```

We spotted my latest commit for MinMaxCalculation.java at the top, making it easy to identify and share.

Always, We ensured to stick to best practices, using descriptive names for branches and commits, and continuously updating our .gitignore file as necessary. This approach not only kept our repo organized but also ensured smooth collaboration with the team.

3. How many commits did you cherry-pick? Are the commit hashes of the cherry-picked commits identical in `main` and `feature-branch`? Briefly explain why.

Hand-selected commits from the feature branch to the main branch totaled 14. The commit hashes for the cherry-picked commits in the feature-branch and main disagree.

When we cherry-pick a commit, Git produces a new commit with a different parent by applying the changes made by the original commit to the current branch. Since the commit hash contains the commit text and its parent(s), among other things, it will differ even if the modifications (diffs) are the same. Because of this, despite communicating exactly the same changes, the hashes between the main and feature-branch differ.

4. What happens if you merge a branch from which you previously cherry-picked single commits? How often do the cherry-picked commits appear in the history? Briefly explain why.

We could run into some repetition when we merge a branch from which you've previously cherry-picked single commits.

Git recognises the changes made by the cherry-picked commit have already been applied to branch B if we cherry-pick a commit from branch A to branch B and subsequently merge branch

A into branch B. Git employs its merging algorithm throughout the merge process to prevent applying the same changes twice.

Regarding how often the history shows the cherry-picked commits:

In the git log history: Each cherry-picked commit will show up in the commit history together with the original commit from the opposite branch. Unless you changed the message during cherry-picking, they will have distinct commit hashes but comparable commit messages.

In the git log diff: The alterations made by the cherry-picked commit will only show once since Git avoids applying the same changes more than once when merging.

This phenomenon is caused by the fact that, even if a commit's content (changes to files, the commit message, etc.) may be the same in both the original and cherry-picked commits, their commit hashes will vary. This is true since the hash additionally considers the parent commit(s) and other data in addition to the text. Git focuses on the changes the commits make rather than the commit hashes themselves when merging. Git will be able to recognise and stop duplicate changes from commits you cherry-picked when you later merge the source branch.

5. What are the risks of rebasing? Mention at least two risks. Briefly describe a use case in which rebasing can be safely applied.

Git's rebasing feature, which enables a linear commit history, is strong yet risky if not utilized wisely. Here are two notable dangers related to rebasing:

Commit History Rewriting: Rebasing's primary function is rewriting the commit history. This might be an issue, particularly if you operate in a team atmosphere. Rebasing pushed commits will result in confusion and conflicts when other people attempt to pull the most recent modifications or push their own if they have based their work on those commits.

Loss of Changes: If rebasing is performed incorrectly, commitments may be lost. For instance, if someone accidentally drops a commit instead of choosing it during an interactive rebase, that modification will be lost. Additionally, resolving disputes during a rebase may sometimes lead to unintentional mistakes or omissions.

Safe Rebasing Use Case:

When working on a feature branch that hasn't been shared with anybody else and attempting to integrate the most recent modifications from the main (or another) branch into your feature branch, rebasing is a frequent and secure technique. This will prevent the need for a merge commit and guarantee that your feature branch is current with the base branch.

A main branch and a feature branch both exist.
Since you diverged with features, new commits have been uploaded to main.

We can rebase features onto main instead of integrating main into it.
This basically incorporates all of the recent commits by moving the whole feature branch to the beginning of the main branch.
You may safely rebase in this circumstance without worrying about disrupting other people's work since the feature branch is private and hasn't been shared with anybody. The feature may be effortlessly merged back into main (or another target branch) once it is finished and the history is clear and linear.

6. What are the risks of using reset when a commit has already been pushed?

There are various dangers associated with using git reset on a commit that has already been pushed, particularly in a collaborative setting:

Conflicting Histories: Git will refuse a push if we undo a previously pushed commit and then attempt to push it again since the histories of the local and remote branches will have changed. While the remote branch will still include the commit(s) we have reset, our local branch will have "gone back in time".

Disrupting Collaborators: After we force-push your modifications, other team members whose branches were dependent on the commit(s) we reset will no longer be in sync with the new remote branch. When they attempt to reconcile the different histories, it may lead to misunderstanding, unnecessary labor, and complicated merge conflicts.

Potential Data Loss: Depending on the reset's mode (--soft, --mixed, or --hard), our staging area or working directory may change. In instance, git reset --hard may erase modifications made to our working directory and staging area. We risk losing work if we reset inadvertently to a commit that is older than we planned or if we don't have a backup of the modifications we reset.

Loss of Commit History: After we force-push our branch and reset it, the remote repository's commit history is overwritten. This makes it more difficult to track down changes and comprehend the development of the codebase since the commit messages, modifications, and any other data related to those commits might be lost.

Given these concerns, it is often advised to utilize alternate techniques that don't change history, such as undoing the commit, particularly in shared branches, if we discover the need to reverse a commit that has been pushed. Always inform our staff before using reset, and be sure to make a backup of any significant changes.

7. Does revert remove the reverted commit? Briefly explain how revert works.

The reversed commit is not deleted or removed from the commit history when we execute git revert. Instead of undoing the modifications made in the commit(s) we're restoring, git revert produces a new commit. This gives us the ability to practically "undo" a commit while keeping a detailed record of the modification in the repository's history.

When you run git revert <commit-hash>, Git identifies the changes introduced in the specific commit.
Git then reverses those modifications, resulting in the reverse set of modifications.
These reversed modifications result in a new commit. When this commit is implemented, the modifications made in the one you're undoing will no longer exist.
All prior commits, including the one you're undoing, are still there and untouched in the commit history, which stays linear.
Because it doesn't modify the commit history, this method makes git reverse a secure way to undo changes, particularly on shared branches. Transparency and traceability are maintained since everyone on the team can see when a commit was made and subsequently revoked.