Version Control (Git)

Introduction to Version Control

- **Version Control Systems (VCSs)** track changes in files and folders.
- Useful for maintaining history and collaboration.
- Snapshot-based tracking of entire directory states.
- Metadata includes authorship, messages, and more.

Why Version Control?

- Track historical snapshots of a project.
- Log changes and reasoning behind them.
- Manage parallel branches of development.
- Resolve conflicts in concurrent development.
- Answer questions about code history.

THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

https://missing.csail.mit.edu/

Understanding Git

- Git's interface can be confusing if approached top-down.
- Learning Git bottom-up, starting with its data model, leads to better understanding.
- Git commands manipulate the underlying data model.

Git’s Data Model

Snapshots

- **Blob**: A file.
- **Tree**: A directory mapping names to blobs or trees.
- **Commit**: A snapshot of the top-level tree with metadata.

```
<root> (tree)
|
+- foo (tree)
|  |
|  + bar.txt (blob, contents = "hello world")
|
+- baz.txt (blob, contents = "git is wonderful")
```

Modeling History: Relating Snapshots

- History in Git is a directed acyclic graph (DAG) of snapshots.
- Each commit refers to a set of parent snapshots.
- Commits are immutable; changes create new commits.

```
o <-- o <-- o <-- o
            ^
             \
              --- o <-- o
```

Objects and Content-Addressing

- Git stores blobs, trees, and commits as objects.
- Objects are content-addressed by their SHA-1 hash.

```
objects = map<string, object>

def store(object):
    id = sha1(object)
    objects[id] = object
```

References

- Human-readable names for SHA-1 hashes.
- Examples: `master`, `HEAD`.
- Mutable pointers to commits.

```
references = map<string, string>

def update_reference(name, id):
    references[name] = id
```

Repositories

- A Git *repository* is a collection of `objects` and `references`.
- Commands manipulate the commit DAG via objects and references.

Staging Area

- Intermediate area to specify modifications for the next commit.
- Allows for clean, selective snapshots.
- Not directly part of the data model, but essential for the interface.

Git Command-Line Interface

Basics

- `git init`: Create a new repo.
- `git status`: Current status.
- `git add <filename>`: Stage files.
- `git commit`: Create a commit.
- `git log`: Show history.

Branching and Merging

- `git branch`: List/create branches.
- `git checkout -b <name>`: Create/switch branches.
- `git merge <revision>`: Merge changes.

Remotes

- `git remote add <name> <url>`: Add remote repo.
- `git push/pull`: Send/receive changes.

Undo

- `git commit --amend`: Edit last commit.
- `git reset HEAD <file>`: Unstage file.
- `git checkout -- <file>`: Discard changes.

Advanced Git

- `git config`: Customize Git settings.
- `git add -p`: Interactive staging.
- `git rebase -i`: Interactive rebasing.
- `git blame`: Show last edit per line.
- `git stash`: Stash changes.
- `git bisect`: Binary search history.

Miscellaneous

- **GUIs**: There are many GUI clients for Git.
- **Shell integration**: Git status in shell prompt.
- **Editor integration**: Git features in text editors.
- **Workflows**: Different practices for using Git.
- **GitHub**: Git hosting with pull requests.
- **Other providers**: GitLab, BitBucket, etc.

Resources

- Pro Git Book
- Oh Shit, Git!?!
- Git for Computer Scientists
- Git from the Bottom Up
- Explain Git in Simple Words
- Learn Git Branching