Version Control (Git)

[Git's data model]

1. Snapshots

- Git models the history of a collection of files and folders within some top-level directory as a series of snapshots. In Git terminology, a file is called a "blob", and it's just bunch of bytes. A directory is called a "tree", and it maps names to blobs or trees (so directories can contain other directories). A snapshot is the top-level tree that is being tracked. An example is like:

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

the top-level tree contains 2 elements: a tree "foo" and a blob "baz.txt"

2. Modeling history: relating snapshots

- In Git, a history is a directed acyclic graph (DAG) of snapshots, which means that each snapshot in git refers to a set of "parents", the snapshots that preceded it. it's a set of parents rather than a single parent, because a snapshot might descend from multiple parents, for example, due to merging/combining two parallel branches of development. Git calls these snapshots "commits".

3. Data model, as pseudocode

```
// a file is a bunch of bytes
type blob = array<byte>

// a directory contains named files and directories
type tree = map<string, tree | blob>

// a commit has parents, metadata, and the top-level tree
type commit = struct {
    parents: array<commit>
    author: string
    message: string
    snapshot: tree
}
```

4. Objects and content-addressing

- An object is a blob, tree or commit

```
type object = blob | tree | commit
```

- In Git data store, all objects are content-addressed by their SHA-1 hash

```
objects = map<string, object>
```

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

def load(id):
    return objects[id]
```

5. References

- SHA-1 hashes is kind of inconvenient. Git's solution to the problem is human-readable names for SHA-1 hashes, called "references". References are pointers to commits. Unlike objects, which are immutable, references are mutable (can be updated to point to a new commit). For example, the master reference usually points to the latest commit in the main branch of development.

```
reference = map<string, string>

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

def read_reference(name):
    return references[name]

def load_reference(name_or_id):
    if name_or_id in references:
        return load(references[name_or_id])
    else:
        return load(name_or_id)
```

With reference, Git can use human-readable names like master to refer to a particular snapshot in the history, instead of a long hexadecimal string.

- In Git, that "we currently are" is a special reference called "HEAD".

6. Repositories

- the definition of a git repository: the data objects and references
- On disk, all git stores are objects and references. All git commands map to some manipulation of the commit DAG by adding object and adding/updating references.

[Staging area]

- Git allows you to specify which modification should be included in the next snapshot through a mechanism called the "staging area". (git add x)

[Git command-line interface]

1. Basics:

git help <command>: get help for a git command

git init: creates a new git repo, with data stored in the .git directory git status: tells you what's going on

git add <filename>: adds files to staging area

git commit: creates a new commit

write good commit messages!

git log: shows a flattened log of history

git log —all —graph —decorate: visualized history as a DAG

git diff <filename>: show changes you made relative to the staging area

git diff <revision> <filename>: shows differences in a file between snapshots

git checkout <revision>: updates HEAD and current branch

2. Branching and merging:

git branch: shows branches

git branch <name>: creates a branch

git branch -M main: set the current branch as main

git checkout -b <name>: creates a branch and switches to it

same as git branch <name>; git checkout <name>

git merge <revision>: merges into current branch

git mergetool: use a fancy tool to help resolve merge conflicts

git rebase: rebase set of patches onto a new base

3. Remotes:

git remote: lists remotes

git remote add <name> <url>: add a remote

git push <remote> <local branch>:<remote branch>: send objects to remote, and update remote reference

git branch —set-upstream-to=remote>/<remote branch>: set up correspondence between local and remote branch

git fetch: retrieves objects/references from a remote

git pull: same as git fetch; git merge

git clone: download repository from remote

4. Undo:

git commit —amend: edit a commit's contents/message

git reset HEAD <file>: unstage a file

git checkout — <file>: discard changes

[Advanced Git]

git config: Git is highly customizable

git clone —depth=1: shallow clone, without entire version history

git add -p: interactive staging

git rebase -i: interactive staging

git blame: show who last edited which line

git stash: temporarily remove modifications to working directory

git bisect: binary search history

.gitignore: specify intentionally untracked files to ignore

[Exercises]

1. Visualize the version history by visualizing it as a graph.

```
git log —all —graph —decorate
```

2. Check the last person to modify README.md

```
git log README.md
```

3. Check the commit message associated with the last modification to the README.md

```
git blame [filename]
->find the hash of the intended line
git show [hash]
```

4. Remove sensitive data from the git history (take my ECE568 as an example)

- Download the git-filter-repo file from the official git repository
- sudo scp git-filter-repo /usr/local/bin (notice that there are no extensions)
- Change the current working directory to another unimportant directory

```
python3 git-filter-repo —analyze
git clone [git repo url]
cd [git repo]
python3 git-filter-repo —invert-paths —path [path of the directory/file to delete]
echo "name of the directory/file to delete" >> .gitignore
git add .gitignore
git commit -m "Trying clear sensitive data from the commit history"
git push origin —force —all
```

5. Clone one repository from Github, and modify one of its existing files. What happens when you do [git stash]? What do you see when running [git log —all —oneline]? Run [git stash pop] to undo what you did with git stash.

git stash

Saved working directory and index state WIP on main: [hash] recover files

git log —all —oneline

Showing a list of {[hash] commits}

I can see the commits history of all branches in a condensed format, showing the commit hash and commit message for each commit.

git stash pop

Git brings the files back to the state I had before using "git stash"

6. git aliases: make [git graph] equivalent to [git log —all —graph —decorate —oneline]

```
vim ~/.gitconfig
G->go the end of file
o->open a new line at the bottom
[alias]
    graph = log —all —graph —decorate —oneline
```