CS 45, Lecture 8
Version Control

Winter 2023

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Outline

1. Review

2. Version Control

3. Git

Outline

1. Review
2. Version Control
3. Git

Computer Networks

Last lecture, we saw:

- How computers can use a network to talk to each other
- How information gets sent from one place to another over the internet

Computer Networks

Last lecture, we saw:

- How computers can use a network to talk to each other
- How information gets sent from one place to another over the internet

In this lecture, we will see:

- How to safely store your files (code or text)
- How to collaborate on files with others over the internet
- How to avoid losing all your homework!

Files

- Many of the files you work with will be text:

Files

- Many of the files you work with will be text:
  - Source Code
  - Documentation
  - Markup Files

Files

- Many of the files you work with will be text:
  - Source Code
  - Documentation
  - Markup Files
- As you change these files over time, you'll eventually want some way to keep track of different “versions” of the file.

Files

- Many of the files you work with will be text:
  - Source Code
  - Documentation
  - Markup Files
- As you change these files over time, you'll eventually want some way to keep track of different “versions” of the file.
- What we need is a “version control system”.

Outline

1. Review

2. Version Control

2.1 Version Control Systems

2.2 Comparison of VCSs

3. Git

Outline

1. Review

2. Version Control

2.1 Version Control Systems

2.2 Comparison of VCSs

3. Git

Version Control Systems

- A VERSION CONTROL SYSTEM (VCS) is a piece of software which manages different versions of your files and folders for you.

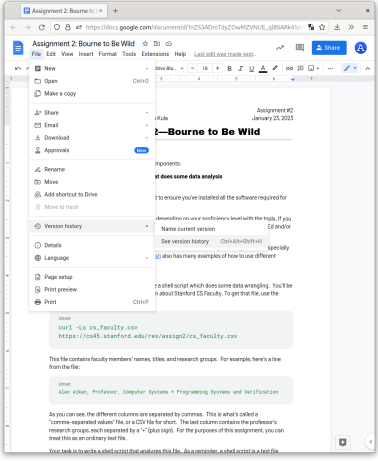
Version Control Systems

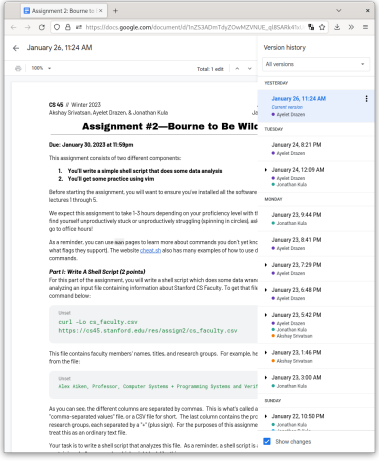
- A VERSION CONTROL SYSTEM (VCS) is a piece of software which manages different versions of your files and folders for you.
- A good VCS will let you look at old versions of files and restore files (or information) which you might have accidentally deleted.

Version Control Systems

- A VERSION CONTROL SYSTEM (VCS) is a piece of software which manages different versions of your files and folders for you.
- A good VCS will let you look at old versions of files and restore files (or information) which you might have accidentally deleted.
- You've seen these before!

Version Control Systems





Version Control Systems

A good version control system:

- Will store many versions of your files

Version Control Systems

A good version control system:

- Will store many versions of your files
- Will let you “revert” a file (or a part of a file) to an older version

Version Control Systems

A good version control system:

- Will store many versions of your files
- Will let you “revert” a file (or a part of a file) to an older version
- Will track the order of different versions

Version Control Systems

A good version control system:

- Will store many versions of your files
- Will let you “revert” a file (or a part of a file) to an older version
- Will track the order of different versions
- Will ensure each “version” is neither too big nor too small

Version Control Systems

A good version control system:

- Will store many versions of your files
- Will let you “revert” a file (or a part of a file) to an older version
- Will track the order of different versions
- Will ensure each “version” is neither too big nor too small

A great version control system:

- Will let you collaborate on files with other people

Version Control Systems

A good version control system:

- Will store many versions of your files
- Will let you “revert” a file (or a part of a file) to an older version
- Will track the order of different versions
- Will ensure each “version” is neither too big nor too small

A great version control system:

- Will let you collaborate on files with other people
- Will combine the different versions of the files produced by different people sanely

Outline

Google Docs

Google Docs automatically keeps track of file history in a basic VCS.

Pros:

- Great for rich text
- Allows real-time collaboration
- Saved on the cloud automatically

Cons:

- Bad for plain text (especially code)
- Requires an internet connection
- Only supports a single “current” version of a single file

Copying Files

You can make a bunch of copies of files or folders with `cp` as a simple form of version control. You can compare versions with `diff`.

Pros:

- Works on either rich or plain text (or anything else)
- It's simple and makes it easy to move data between versions

Cons:

- It's messy and a lot of manual work
- It's hard to tell what the relationship between different versions is
- It takes a lot of hard drive space

Zip Files

Instead of just `cp`ing folders, we could bundle them up into a Zip file (a single file which can be “unzipped” into a folder).

Pros:

- Tracks versions for an entire folder at once
- Easy to share a version with someone else (email)

Cons:

- It's still a lot of manual work
- It's hard to tell what the relationship between different versions is
- It's hard to extract a single file from an old version

Zip Files++

- What if we had a tool which did all this zip file stuff automatically?

Zip Files++

- What if we had a tool which did all this zip file stuff automatically?
- We could tell it to take a “snapshot” of a directory, and it would save all the changes in it.

Zip Files++

- What if we had a tool which did all this zip file stuff automatically?
- We could tell it to take a “snapshot” of a directory, and it would save all the changes in it.
- We could ask it to recover an old version of a specific file, or to reset everything to an old version to “undo” our work.

Zip Files++

- What if we had a tool which did all this zip file stuff automatically?
- We could tell it to take a “snapshot” of a directory, and it would save all the changes in it.
- We could ask it to recover an old version of a specific file, or to reset everything to an old version to “undo” our work.
- The tool could track the relationships between different versions, so we can have multiple “current” versions at the same time.

Zip Files++

- What if we had a tool which did all this zip file stuff automatically?
- We could tell it to take a “snapshot” of a directory, and it would save all the changes in it.
- We could ask it to recover an old version of a specific file, or to reset everything to an old version to “undo” our work.
- The tool could track the relationships between different versions, so we can have multiple “current” versions at the same time.
- If we want to combine different versions, the tool can automatically do it for us (instead of us copying and pasting the parts together).

Git

`git` is a version control system which tracks “commits” (snapshots) of files in a REPOSITORY.

Git

`git` is a version control system which tracks “commits” (snapshots) of files in a REPOSITORY.

- Git stores old versions of files in a hidden folder (`.git`), and automatically manages them.

Git

`git` is a version control system which tracks "commits" (snapshots) of files in a REPOSITORY.

- Git stores old versions of files in a hidden folder (`.git`), and automatically manages them.
- We can tell Git to keep track of certain files, and tell it when to take a snapshot.

Git

`git` is a version control system which tracks "commits" (snapshots) of files in a REPOSITORY.

- Git stores old versions of files in a hidden folder (`.git`), and automatically manages them.
- We can tell Git to keep track of certain files, and tell it when to take a snapshot.
- We can ask Git to go back to an old snapshot (even for a single file).

Git

`git` is a version control system which tracks “commits” (snapshots) of files in a REPOSITORY.

- Git stores old versions of files in a hidden folder (`.git`), and automatically manages them.
- We can tell Git to keep track of certain files, and tell it when to take a snapshot.
- We can ask Git to go back to an old snapshot (even for a single file).
- We can ask Git to keep track of who's working on what, so multiple people can work on different things without conflicting.

Git

`git` is a version control system which tracks “commits” (snapshots) of files in a REPOSITORY.

- Git stores old versions of files in a hidden folder (`.git`), and automatically manages them.
- We can tell Git to keep track of certain files, and tell it when to take a snapshot.
- We can ask Git to go back to an old snapshot (even for a single file).
- We can ask Git to keep track of who's working on what, so multiple people can work on different things without conflicting.
- If we want to combine multiple people's work, we can ask Git to automatically merge them together. If it can't for some reason, it'll ask us to manually merge them.

Outline

1. Review

2. Version Control

3. Git

3.1 Linear History

3.2 Branching Workflow

3.3 Combining Branches

Outline

1. Review

2. Version Control

3. Git

3.1 Linear History

3.2 Branching Workflow

3.3 Combining Branches

Basic Workflow

The simplest way to use git is the “linear” workflow, which is the same way you'd use Google Docs:

Basic Workflow

The simplest way to use git is the “linear” workflow, which is the same way you'd use Google Docs:

1. `git init` to enable Git in a certain directory

Basic Workflow

The simplest way to use git is the “linear” workflow, which is the same way you'd use Google Docs:

1. `git init` to enable Git in a certain directory
2. `git add` any files you want Git to “track”

Basic Workflow

The simplest way to use git is the “linear” workflow, which is the same way you'd use Google Docs:

1. `git init` to enable Git in a certain directory
2. `git add` any files you want Git to “track”
3. `git commit` the currently “staged” changes to save a snapshot

Basic Workflow

The simplest way to use git is the “linear” workflow, which is the same way you'd use Google Docs:

1. `git init` to enable Git in a certain directory
2. `git add` any files you want Git to “track”
3. `git commit` the currently “staged” changes to save a snapshot
4. make changes to your files

Basic Workflow

The simplest way to use git is the “linear” workflow, which is the same way you'd use Google Docs:

1. `git init` to enable Git in a certain directory
2. `git add` any files you want Git to “track”
3. `git commit` the currently “staged” changes to save a snapshot
4. make changes to your files
5. `git add` the changed files to “stage” them again

Basic Workflow

The simplest way to use git is the “linear” workflow, which is the same way you'd use Google Docs:

1. `git init` to enable Git in a certain directory
2. `git add` any files you want Git to “track”
3. `git commit` the currently “staged” changes to save a snapshot
4. make changes to your files
5. `git add` the changed files to “stage” them again
6. Repeat from 3

You can use `git log` to see your commit history, and use `git status` to see the current state of staged/unstaged/untracked changes.

Basic Workflow

Demo

Let's practice how to:

- Create a new Git repository
- Commit a new file
- Commit changes to files
- Revert commits
- Look at an old version of a file
- Compare two versions of files
- See your commit history

Outline

Branching Workflow

We can also split our “repo” into multiple BRANCHES, which are like alternate versions of a folder. This means different people can work on different things without interfering with one another.

Branching Workflow

We can also split our “repo” into multiple BRANCHES, which are like alternate versions of a folder. This means different people can work on different things without interfering with one another.

1. Make sure your repository is “clean” (i.e., you have no uncommitted changes).

Branching Workflow

We can also split our “repo” into multiple BRANCHES, which are like alternate versions of a folder. This means different people can work on different things without interfering with one another.

1. Make sure your repository is “clean” (i.e., you have no uncommitted changes).
2. `git checkout -b <branch>` to create a new branch and move to it; at this point, the new branch will be identical to the old one.
3. Make changes, `git add`, `git commit` as usual
4. `git checkout` to switch between branches

Outline

1. Review

2. Version Control

3. Git

3.1 Linear History

3.2 Branching Workflow

3.3 Combining Branches

Branching Workflow

Combining Branches

Now that we have multiple branches, we probably want to join them back together at some point.

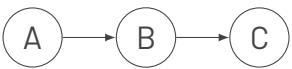
There are several ways to do this:

- `git merge` two branches into one
- `git merge --fast-forward` a long branch onto a shorter version of itself
- `git rebase` one branch onto another branch
- `git cherry-pick` a specific commit from one branch to another

Branching Workflow

Fast Forwarding

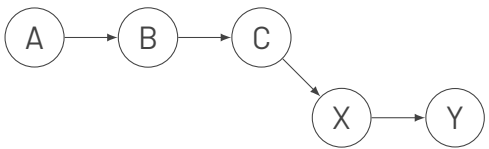
The simplest case of MERGING is called FAST-FORWARDING.



Branching Workflow

Fast Forwarding

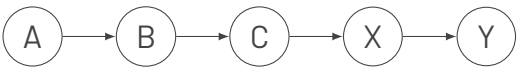
The simplest case of MERGING is called FAST-FORWARDING.



Branching Workflow

Fast Forwarding

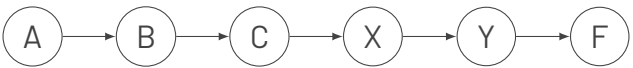
The simplest case of MERGING is called FAST-FORWARDING.



Branching Workflow

Fast Forwarding

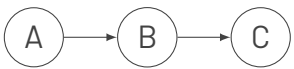
The simplest case of MERGING is called FAST-FORWARDING.



Branching Workflow

Merging

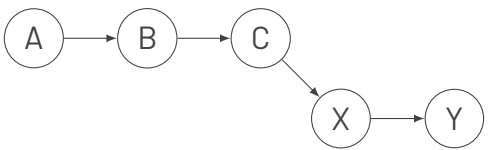
MERGING (in general) creates a MERGE COMMIT to join the two branches.



Branching Workflow

Merging

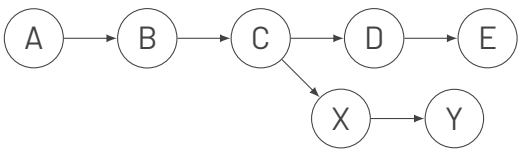
MERGING (in general) creates a MERGE COMMIT to join the two branches.



Branching Workflow

Merging

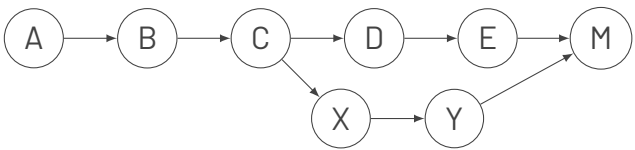
MERGING (in general) creates a MERGE COMMIT to join the two branches.



Branching Workflow

Merging

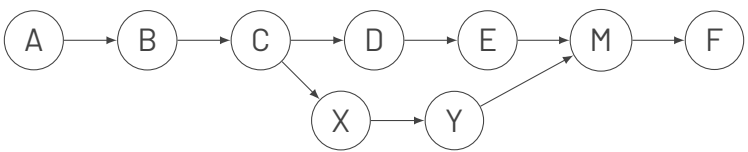
MERGING (in general) creates a MERGE COMMIT to join the two branches.



Branching Workflow

Merging

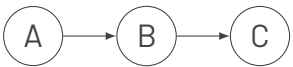
MERGING (in general) creates a MERGE COMMIT to join the two branches.



Branching Workflow

Rebasing

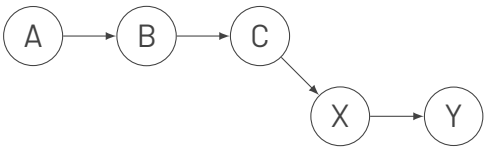
REBASING moves the “base” of a branch to be a different commit.



Branching Workflow

Rebasing

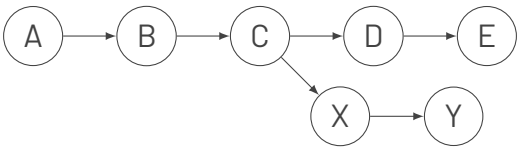
REBASING moves the “base” of a branch to be a different commit.



Branching Workflow

Rebasing

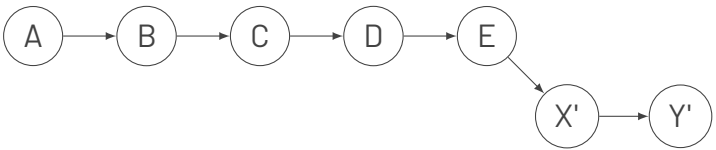
REBASING moves the “base” of a branch to be a different commit.



Branching Workflow

Rebasing

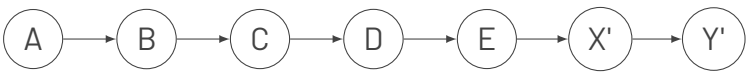
REBASING moves the “base” of a branch to be a different commit.



Branching Workflow

Rebasing

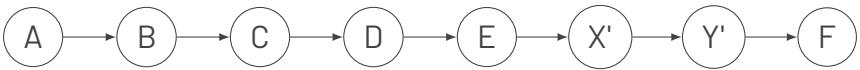
REBASING moves the “base” of a branch to be a different commit.



Branching Workflow

Rebasing

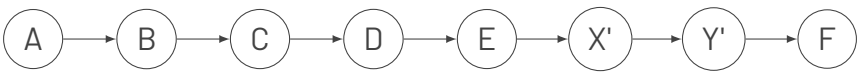
REBASING moves the “base” of a branch to be a different commit.



Branching Workflow

Rebasing

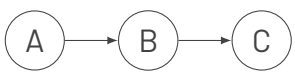
REBASING moves the “base” of a branch to be a different commit. REBASING edits Git's history to make FAST-FORWARDING possible.



Branching Workflow

Cherry-Picking

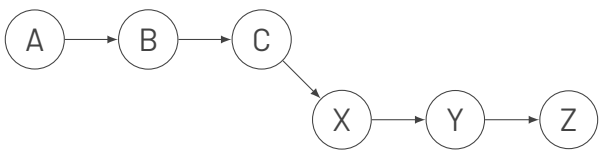
CHERRY-PICKING copies a *single commit* from one branch to another branch.



Branching Workflow

Cherry-Picking

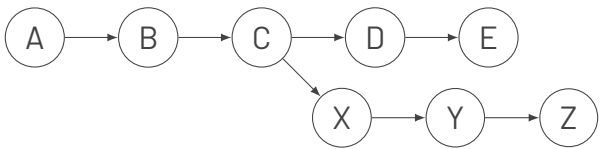
CHERRY-PICKING copies a *single commit* from one branch to another branch.



Branching Workflow

Cherry-Picking

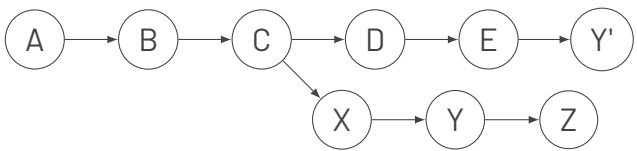
CHERRY-PICKING copies a *single commit* from one branch to another branch.



Branching Workflow

Cherry-Picking

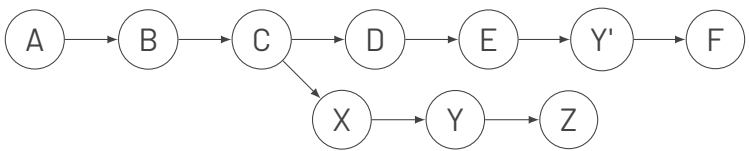
CHERRY-PICKING copies a *single commit* from one branch to another branch.



Branching Workflow

Cherry-Picking

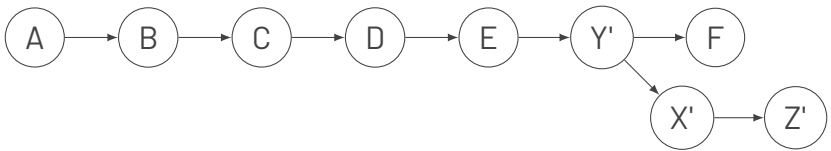
CHERRY-PICKING copies a *single commit* from one branch to another branch.



Branching Workflow

Cherry-Picking

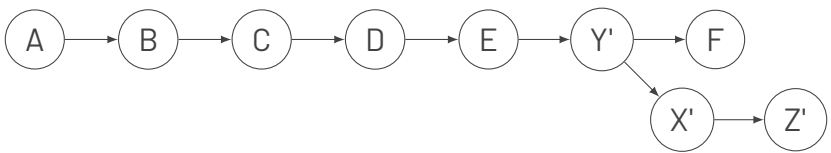
CHERRY-PICKING copies a *single commit* from one branch to another branch.



Branching Workflow

Cherry-Picking

CHERRY-PICKING copies a *single commit* from one branch to another branch. CHERRY-PICKING and rebasing is a good way to move a single commit from one branch to another.



Branching Workflow

When to merge/rebase/cherry-pick?

- **fast-forward** when possible (`git merge --ff-only`).
- **rebase and then fast-forward** if possible, i.e., if you're the only one working on the branch; **never** rebase a branch other people are using (`git rebase` and `git merge --ff-only`).
- **merge** if neither of the above are possible (`git merge`).
- **cherry-pick** if you want to copy a specific commit to another branch (`git cherry-pick`)[1].

[1] This is pretty rare, I've only used it a handful of times.

Branching Workflow

Branching Demo

Let's practice how to:

- Split our repository into two branches
- Switch between branches
- Make commits on either branch
- Merge two branches together

To Be Continued...

We'll pick back up with merge conflict resolution and collaboration in Lecture 9.

Some commands which came up during class:

`git checkout:` essentially means “move to a different commit”; doesn't change your git history

`git reset:` “resets” the entire repository to the way it was in an old commit (and changes git history to match)

`git revert:` “undoes” a specific old commit by creating a new commit that does the opposite

Note that, even though Git commits are technically versions, Git's commands often operate on the *changes* between versions.