Missing semester: ANSWERS

Version Control (Git)

Theory:

1) Git is a **version control system** (VCS). VCSs are tools used to **track changes to source code** (or others collections of files and folders).
2) In Git, a history is a **directed acyclic graph** (DAG) of snapshots. All this means is that each snapshot in Git refers to **a set of "parents"**, the snapshots that preceded it.
3) In Git, typically a tree is a **folder** and a blob is a **file**.
4) In Git data store, all objects are content-addressed by their **SHA-1 hash**.
5) In Git, a reference is a **pointer to a commit**. More specifically, references are human-readable **names for SHA-1 hashes**. For example, the ***master*** reference usually points to the latest commit in the main branch of development, and the ***HEAD*** is a special reference for "where we currently are".
6) In Git, rebasing and merging are both designed to integrate changes from one branch into another branch but in different ways. Suppose you have two branches *master* and *features* and you want to integrate the changes from *features* to *master*. If you merge, you take the contents of the *feature* branch and integrate it with the *master* branch within a new commit. If you rebase the *feature* branch into *master*, you move the base of the feature branch to the *master* branch's ending point. In conclusion, **merge preserves history whereas rebase rewrites it**.
7) In Git, cloning a remote repository gives you a local copy of it and sets up **all the additional remote tracking branches**, whereas pulling a remote repository will only give you a local copy of a particular branch. In practice, you clone a remote repository only once and the first time you want to work on it and then you pull to **update that local copy with new commits** from the remote repository.
8) In Git, fetching and pulling are both used to pull new changes from a remote branch into a local branch. Whereas pulling incorporates those changes onto your target branch (for the purpose to update it), fetching only stores them in a new branch in your local repository. Basically, **pulling = fetching + merging**.
9) 20+ Git commands (without options) along with one line explanation:
    - git help <command>: get help for a Git command.
    - git init: creates a new Git repository, with data stored in the .git directory.
    - git status: tells you a bunch of information about what's going on.
    - git add <filename>: adds files to the staging area.
    - git commit: creates a new commit.
    - git log: shows a flattened log of history.
    - git diff <filename>: show changes 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.
    - git branch: shows branches.

- git branch <name>: creates a branch.
- git merge <revision>: merges into current branch.
- git rebase: rebase set of patches onto a new base.
- git remote: list 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 fetch: retrieve objects/references from a remote.
- git pull: same as git fetch; git merge.
- git clone: download repository from remote.
- git commit --amend: edit a commit's contents/message.
- git reset HEAD <file>: unstage a file.
- git blame: show who last edited which line.
- git stash: temporarily remove modifications to working directory.
- git bisect: binary search history (e.g. for regressions).

10) One of the seven commonly accepted rules on how to write a Git commit is to limit the subject line to **50 characters**.

Practice:

1) a) git log --all --graph --decorate --oneline
   b) git log README.md
   c) git blame -L 7,8 README.md
      git show {SHA-1 hash}
2) git checkout master
   git reset --hard 5d83f9e
3) git diff-tree -r 5d83f9e
4) Use the command *git stash* to save your local modifications away and run *git stash pop* to come back to your work.
5) git filter-branch
   Bonus: git filter-branch --force --index-filter "git rm --cached --ignore-unmatch <filepath>" --prune-empty --tag-name-filter cat -- --all
6) echo "*.txt" > .gitignore_global
   echo "*.o" >> .gitignore_global
   git config --global core.excludesfile ~/.gitignore_global
7) With the command *git revert*.
8) git reset <filename>
9) git reset –soft HEAD~N &&git commit
10) Git bisect uses a binary search algorithm to find which commit in your project's history introduced a bug. You use it by first telling it a "bad" commit that is known to contain the bug, and a "good" commit that is known to be before the bug was introduced. Then Git bisect picks a commit between those two endpoints and asks you whether the selected commit is "good" or "bad". It continues narrowing down the range until it finds the exact commit that introduced the change.

Debugging and Profiling

Theory:

1) In UNIX systems, it is commonplace for programs to write their logs under **/var/log**.
2) The *journalctl* bash command displays the log messages of **systemd**, the system log of Linux, which can be found in the folder **/var/log/journal**. Similarly, on macOS, the ***log show*** command displays the system log messages, which can be found in the **/var/log/system.log** folder.
3) In UNIX systems terminology, a logger is a **shell program** that is used for **logging under the system logs**.
4) The command ***logger "Hello World"*** will display "Hello World" in the system log. To see it, use the command ***log show --last 1m | grep Hello*** on macOS and ***journalctl --since "1m ago" | grep Hello*** on Linux.
5) In Python, the **Python Debugger (pdb)** is the commonly used python debugger.
6) 5+ pdb commands along with one line explanation:
    - **l**: For list. Displays 11 lines around the current line or continue the previous listing.
    - **s**: For step. Execute the current line, stop at the first possible occasion.
    - **n**: For next. Continue execution until the next line in the current function is reached or it returns.
    - **b**: For break. Set a breakpoint (depending on the argument provided).
    - **p**: For print. Evaluate the expression in the current context and print its value.
    - **r**: For return. Continue execution until the current function returns.
    - **q**: For quit. Quit the debugger.
7) A profiler is a tool that aims to aid **program optimization**. For example, a profiler measures the **space memory** or **time complexity** of a program, the **usage** of particular instructions, or the **frequency** and **duration** of function calls.
8) It is common for tools to make a distinction between **Real time**, **User time** and **Sys time**. The Real time measures the **elapsed time from start to finish** of the program, including the time taken by other processes and time taken while blocked (e.g. waiting for I/O or network). The User time is the amount of time spent in the **CPU running user code**.The Sys time is the amount of time spent in the **CPU running kernel code**.
9) In languages like C or C++ memory leaks can cause your program to never release memory that it doesn't need anymore. To help in the process of memory debugging you can use tools like **Valgrind** that will help you identify memory leaks.
10) One common way to display CPU profiling information for sampling profilers is to use a **Flame Graph**, which will display a hierarchy of function calls across the Y axis and time taken proportional to the X axis.

Metaprogramming

Theory:

1) To define a build process for build systems, one must define a number of **dependencies**, a number of **targets**, and **rules** for going from one to the other.
2) **make** is one of the most common build systems. When you run the command *make*, it consults a file called **Makefile** in the current directory. All the targets, their dependencies, and the rules are defined in that file, like this:
```
target: dependency1 dependency2
	rule1
	rule2
```
   Note that there is a tab indent for each of the rules (and no 4 spaces).
3) 3 examples of patterns for build process along with one line explanation:
   - **%**: **wildcard** that can match any nonempty substring. Match the same string on the left and on the right in a *target: dependency* relation.
   - **$***: special variable which gets replaced by the **stem** -the part of the target that was matched by the pattern rule - with which the rule matched.
   - **$@**: special variable which gets replaced by the **name of the target** being generated.
4) **Apt** for Ubuntu system packages, **PyPi** for Python libraries and **RubyGems** for Ruby libraries are examples of repositories that host a large number of dependencies.
5) With semantic versioning, every version number is of the form: **major.minor.patch**.
6) In semantic versioning, the rules are:
   - If a new release does not change the API, increase the patch version.
   - If you add to your API in a backwards-compatible way, increase the minor version.
   - If you change the API in a non-backwards-compatible way, increase the major version.
7) A lock file is simply a file that **lists the exact version** you are currently depending on of each dependency. There are many reasons for this, such as **avoiding unnecessary recompiles**, **having reproducible builds**, or **not automatically updating to the latest version** (which may be broken).
8) Continuous integration, or CI, is an umbrella term for "**stuff that runs whenever your code changes**". As an example of a CI system, Metadot websites are set up using GitHub Pages.
9) A test suite is, as its name implies, a **suite of tests**. It's a collective term for all the tests. For example, there are **unit test** - a "micro-test" that tests a specific feature in isolation, **integration test** - a "macro-test" that runs a larger part of the system to check that different feature or components work *together*, and **regression test** - a test that implements a particular pattern that *previously* caused a bug to ensure that the bug does not resurface.
10) In test terminology, mocking is replacing a function, module, or type with a fake implementation to avoid testing unrelated functionality.

Security and Cryptography

Theory:

1) Entropy is a **measure of randomness**. It is useful, for example, when determining the strength of a password. Entropy is measured in bits, and when selecting uniformly at random from a set of possible outcomes Ω, the entropy is equal to **log2(#Ω)**.
2) A cryptographic hash function maps data of arbitrary size to a fixed size. The output of the SHA1 hash function is a **160-bit** hash value, often represented as **40 hexadecimal characters**.
3) 3+ properties of hash functions along with their meaning:
    - Deterministic: the same input always generates the same output.
    - Non-invertible: it is hard to find an input m such that hash(m) = h for some desired output h.
    - Target collision resistant: given an input m1, it's hard to find a different input m2 such that hash(m1) = hash(m2).
    - Collision resistant: it's hard to find two inputs m1 and m2 such that hash(m1) = hash(m2) (note that this is a strictly stronger property than target collision resistance).
4) Hash functions are used in **Git** to address its objects and by **web hosting softwares** to check that a software downloaded is trustworthy (e.g. identical to the original) by comparing its hash output with the one from the official site.
5) In cryptography, a key derivation function (KDF) is a **hash function** that derives one or more secret keys from a secret value such as a main key, a password, or a passphrase using a **pseudorandom function**. KDFs can be used to **stretch keys into longer keys** or to **obtain keys of a required format**.
6) Symmetric cryptography is constituted of 3 elements:
    - A key generation function that produces a key: *keygen() -> key*.
    - An encrypt function: *encrypt(plaintext, key) -> ciphertext*.
    - A decrypt function: *decrypt(ciphertext, key) -> plaintext*.
7) Asymmetric cryptography is constituted of 3 elements:
    - A key generation function that produces a pair of keys: *keygen() -> (public-key, private-key)*.
    - An encrypt function: *encrypt(plaintext, public-key) -> ciphertext*.
    - A decrypt function: *decrypt(ciphertext, private-key) -> plaintext*.
8) When you run the command *ssh-keygen*, it generates (randomly) an **asymmetric keypair**: *(public-key, private-key)*. The public key is stored as-is, but at rest, the private key should be **encrypted on disk**. The ssh-keygen program prompts the user for a **passphrase**, and this is fed through a **key derivation function** to produce a key, which is then used to encrypt the private key with a **symmetric cipher**.
9) To encrypt its full disk, one can use **cryptsetup** on Linux, **Bitlocker** on Windows, or **FireVault** on macOS.
10) **KeePassXC**, **pass**, and **1Password** are three examples of password managers.

Practice:

1) The cardinality of the set of possible outcomes Ω is equal to: $\#\Omega = 100\ 000^4 = 10^{20}$.
   Because the worlds are selected uniformly at random from this set of possible outcomes Ω, the entropy H is equal to: $H = log_2(\#\Omega) = log_2(10^{20}) = 66.4$ bits.
2) The cardinality of the set of possible outcomes Ω is equal to: $\#\Omega = (26 * 2 + 9)^8 = 61^8 = 1.92 * 10^{14}$.
   Because the worlds are selected uniformly at random from this set of possible outcomes Ω, the entropy H is equal to: $H = log_2(\#\Omega) = log_2(1.92 * 10^{14}) = 47.4$ bits.
3) The attacker can try guessing $10\ 000 * 60 * 60 * 24 = 8.64 * 10^8$ passwords per day.
   Then, on average, to break them, it will take $2^{66.4}/(8.64 * 10^8) = 1.13 * 10^{11}$ days, or 300 millions of centuries for the first password and $2^{47.4}/(8.64 * 10^8) = 2.15 * 10^5$ days, or 6 centuries for the second password.
4) printf 'hello' | sha1sum
5) To check if the file is trustworthy, I will cross-check the hash of the downloaded file using the command *sha256sum debian-10.7.0-amd64-netinst.iso* with the hash retrieved from the official website of the original file.
6) To encrypt the file: *openssl aes-256-cbc -salt -in {input filename} -out {output filename}*.
   To decrypt the file: *openssl aes-256-cbc -d -in {input filename} -out {output filename}*.
7) We can combine the idea of symmetric cryptography with key derivation functions: one can use a passphrase to pass it through the key derivation function to obtain a key and then use this key for the symmetric cryptography. Now we don't need anymore to save the last key but just the passphrase, and whenever we need the key, we can reconstruct it again with the key derivation function and the passphrase.
8) To sign a git commit: *git commit -S*.
   To verify the signature on the commit: *git show --show-signature*.
9) To create a signed tag: *git tag -s*.
   To verify the signature on the tag: *git tag -v*.
10) One can use **GPG** for encrypting messages with asymmetric cryptography.
    To encrypt a message: *gpg2 --armor --encrypt --recipient john@doe.com message.txt*.
    To decrypt a message: *gpg2 --decrypt message.txt.asc > message.txt*.