Assignment 6. Git repository organization

Useful pointers

- Scott Chacon and Ben Straub, Pro Git
- Linus Torvalds, Jun Hamano *et al.*, Git - local branching on the cheap

Homework: Topologically ordered commits

Given a git repository, the commits can be thought of as having the structure of a directed acyclic graph (DAG) with the commits being the vertices. In particular, one can create a directed edge from each child commit to each of its parent commits. Alternatively, one can create a directed edge from each parent to each of its children. Note that if a commit is a merge commit, it will have two or even more parents. In that case, one has to consider all parents.

Follow these five steps and implement `topo_order_commits.py` using `/usr/local/cs/bin/python3` on the SEASnet GNU/Linux servers.

1. *Discover the `.git` directory.* Inside a directory, when the script `topo_order_commits.py` (which doesn't have to reside in the same directory) is invoked, the script should first determine where the top level Git directory is. The top level Git directory is the one containing the `.git` directory. One can do this by looking for `.git` in the current directory, and if it doesn't exist search the parent directory, etc. This discovery process should only go up, and never descend into a child directory. Output a diagnostic '`Not inside a Git repository`' to standard error and exit with status 1 if `.git` cannot be found when the search went all the way to the `/` directory.
2. *Get the list of local branch names.* Figure out what the different directories inside `.git` do, particularly the refs and objects directories. Beware of branch names with forward slashes. Read §10.2 Git Internals – Git Objects and §10.3 Git Internals – Git References. One can use the `zlib` library in Python to decompress Git objects. To simplify this assignment, you can assume that the repository does not use packfiles (see §10.4 Git Internals – Packfiles), i.e., that all its objects are loose.
3. *Build the commit graph.* Each commit can be represented as an instance of the CommitNode class, which you can define as follows, and which you are also free to modify or not use at all:

```
class CommitNode:
    def __init__(self, commit_hash):
        """
        :type commit_hash: str
        """
        self.commit_hash = commit_hash
        self.parents = set()
        self.children = set()
```

The commit graph consists of all the commit nodes from all the branches. Each commit node might have multiple parents and children.

In particular, for each branch, perform a depth-first search traversal starting from the branch head (i.e., the commit pointed to by the branch), to establish the parent-child relationships between the commit nodes. The traversal should trace through the parents, and for every possible pair of parent and child, add the child hash to the parent node's children, and add the parent hash to the child node's parents. The leaf nodes for each branch will be the root commits for that branch, where the leaf nodes are the nodes without any parents. Let `root_commits` be the union of all the leaf nodes across all the branches. If a commit is not reachable from any of the branch heads, it should not be part of the commit graph.

4. *Generate a topological ordering of the commits in the graph.* A topological ordering for our case would be a total ordering of the commit nodes such that all the descendent commit nodes are strictly less than the ancestral commits, where nodes in `root_commits` are the oldest ancestors. One way to generate a topological ordering is to use a depth-first search; see Topological sorting.

5. *Print the commit hashes in the order generated by the previous step, from the least to the greatest.* If the next commit to be printed is not the parent of the current commit, insert a "sticky end" followed by an empty line before printing the next commit. The "sticky end" will contain the commit hashes of the parents of the current commit, with an equal sign "=" appended to the last hash. These hashes of the parents, if any, can be printed in any order separated by whitespace. If there are no parents, just print an equal sign, "=". This is to inform us how the fragments can be "glued" together.

On the other hand, if an empty line has just been printed, before printing the first commit C in a new segment, print a "sticky start" line starting with an equal sign, "=", followed by the hashes of the children of C, if any, on the same line in any order and separated by whitespace, so that we know which commits led to commit C. There is no whitespace after the equal sign.

Furthermore, if a commit corresponds to a branch head or heads, the branch names should be listed after the commit in lexicographical order, separated by white space. Note that this rule does not apply to the hashes in the sticky starts or sticky ends.

The commit hashes in the sticky starts and sticky ends are not considered as part of the sequence of topologically ordered commit output. They are printed only as a visual guide. So even if a commit hash has already appeared in a sticky start or sticky end, it still must be printed as part of the normal sequence of topologically ordered commit output.

Example 1. Consider the following commits where c3 is a child of c1, and c5 is a child of c4:

```
c0 -> c1 -> c2 (branch-1)
       \
        c3 -> c4 (branch-2, branch-5)
               \
                c5 (branch-3)
```

A valid topological ordering from the least to the greatest will be (c5, c4, c3, c2, c1, c0) which should give the following output (assuming the commit hash for cX is hX, and the triple grave accents (```) are not part of the output but merely indicate the start and end of the output lines):

```
```
h5 branch-3
h4 branch-2 branch-5
h3
h1=

=
h2 branch-1
```
```

~~~
h1
h0
```
~~~

A equally valid topological ordering from the least to the greates will be (c2, c5, c4, c3, c1, c0), which should give the following output:

~~~
```

h2 branch-1
h1=


=
h5 branch-3
h4 branch-2 branch-5
h3
h1
h0
```
~~~

Example 2. For the following commits where c6 is a merge commit with parents c2 and c4:

```
c0 -> c1 -> c2 -> c6 (branch-1)
       \         /
        c3 -> c4
```

A topological ordering is (c6, c2, c4, c3, c1, c0), and the corresponding output should be:

~~~
```

h6 branch-1
h2
h1=


=h6
h4
~~~

~~~
h3
h1
h0
```
~~~

Implementation notes.

1. Do not invoke any Git commands in any way. For example, do not use Python subprocesses to call Git. Come to think of it, do not invoke any other commands either; just stick to Python. Use `strace` to verify that your implementation does not invoke commands.
2. The output of `topo_order_commits.py` should be deterministic. For a given Git repository, even though there might be multiple possible ways to perform DFS, multiple valid topological orderings and multiple valid outputs, different invocations of `topo_order_commits.py` should produce identical outputs. For example, since the iteration order on a set `s` in Python is not deterministic, one way to ensure determinism is to call `sorted(s)` which will give a sorted list.
3. Use only the modules in the Python Standard Library. In fact, `os`, `sys`, and `zlib` are the only libraries you need.
4. Points might be deducted if the naming of local variables and functions are not in snake case.

Testing. Please use our test suite to make sure you are on the right track. The `README` file contains instructions on how to use it. Note that passing all the test cases in the test suite does not guarantee a completely correct implementation.

Submit

Submit the file `topo_order_commits.py`. Put commentary in the homework as Python comments. Your commentary should mention how you used `strace` to verify that your implementation does not use other commands.