Dealing with remaining commits

Now we just have to consider what to do about the additional commits on `master`. This is going to depend on a few things. Do you have additional commits on `master` that you haven't distributed among new branches? Do you want `master` to exactly reflect the history of `master` on `upstream`? How tolerant are you of "messy history"? It's also going to depend on whether `master` had been pushed up and whether others might have built on top of it.

There are many options, here are two:

1. Revert the commits that you "moved" to the branches. `git revert` creates "opposite" commits, that undo any changes. So if you lay a revert commit on top of a commit then the content is as though the commit was never made.
2. Using `git reset -hard`, create a new branch from your branching point, then rename it to `master`, orphaning your current master.

Using revert

First I created a copy of the repo, `practice_split_revert/`, so I could show both options. Then I switched to the `master` branch in that copy.

```
cd ..
cp -r practice_split practice_split_revert
cd practice_split_revert/
git branch
```

```
* first_set_branch
  master
  second_set_branch
```

```
git checkout master
```

```
Switched to branch 'master'
```

```
git branch
  first_set_branch
* master
  second_set_branch
git log --oneline --abbrev-commit --all --graph --decorate --color
```

```
* 0841579 (second_set_branch) added to second set
* 0d8e90d first edit in second set
| * 9f2b69a (first_set_branch) added second file to first set
| * ef2a3ee first set edit
|/
| * b5d4aff (HEAD -> master) added to second set
| * dfa98b0 first edit in second set
| * 43bc368 added second file to first set
| * 51f2622 first set edit
```

```
|/
* f945961 add master2
* 367e25a Adding readme
```

Then I used a range expression (the `..`) with revert to revert the set of four commits on `master`. Note that the double dot range syntax used like this is left exclusive (i.e. you have to go back one further than the first you want). I also had to provide four commit messages, one for each of the reverting commits.

```
git revert f945961..b5d4aff
[master afcee46] Revert "added to second set"
 1 file changed, 1 deletion(-)
 delete mode 100644 second_set_file2
[master 45731a7] Revert "first edit in second set"
 1 file changed, 1 deletion(-)
 delete mode 100644 second_set_file
[master 302d9de] Revert "added second file to first set"
 1 file changed, 1 deletion(-)
 delete mode 100644 first_set_file2
[master 0e022e3] Revert "first set edit"
 1 file changed, 1 deletion(-)
 delete mode 100644 first_set_file
git log --oneline --abbrev-commit --all --graph --decorate --color
* 0e022e3 (HEAD -> master) Revert "first set edit"
* 302d9de Revert "added second file to first set"
* 45731a7 Revert "first edit in second set"
* afcee46 Revert "added to second set"
* b5d4aff added to second set
* dfa98b0 first edit in second set
* 43bc368 added second file to first set
* 51f2622 first set edit
| * 0841579 (second_set_branch) added to second set
| * 0d8e90d first edit in second set
|/
| * 9f2b69a (first_set_branch) added second file to first set
| * ef2a3ee first set edit
|/
* f945961 add master2
* 367e25a Adding readme
```

```
ls
```

```
README
```

git reset –hard

Again, I create a copy of the repo, practice_split_reset_master:

```
cp -r practice_split practice_split_reset_master
cd practice_split_reset_master/
git checkout master
```

```
Switched to branch 'master'
```

```
git log --oneline --abbrev-commit --all --graph --decorate --color
```

```
* 0841579 (second_set_branch) added to second set
* 0d8e90d first edit in second set
| * 9f2b69a (first_set_branch) added second file to first set
| * ef2a3ee first set edit
|/
| * b5d4aff (HEAD -> master) added to second set
| * dfa98b0 first edit in second set
| * 43bc368 added second file to first set
| * 51f2622 first set edit
|/
* f945961 add master2
* 367e25a Adding readme
```

Now we use `git reset --hard` to force HEAD back to `f945961`. That disconnects the four unwanted commits (I *think* they aren't actually gone from .git folder yet, but they could be deleted any time as they aren't connected to anything.).

```
git reset --hard f945961
```

```
HEAD is now at f945961 add master2
```

```
git log --oneline --abbrev-commit --all --graph --decorate --color
```

```
* 0841579 (second_set_branch) added to second set
* 0d8e90d first edit in second set
| * 9f2b69a (first_set_branch) added second file to first set
| * ef2a3ee first set edit
|/
* f945961 (HEAD -> master) add master2
* 367e25a Adding readme
```

That seems neater, but keep in mind that if you'd pushed `master` to github while those commits were there, anyone else who had built on your repo would have lots of trouble. The `revert` approach goes for messier history, but has the advantage that it doesn't disconnect anyone else.

Rebase for synchronizing work

When one is working on a problem, others may be working in parallel. And their parallel work may be finished before one's own work is. Thus the starting point for one person's work (branching point) can "go stale" making it harder to integrate.

While `git merge` and resolving syntax level conflicts can resolve some of this, it is often easier to understand and review work if it is presented as changes against an updated starting point.

As a concrete example imagine a project to build a dashboard. Imagine that you fork the repo in January to implement a new type of visualization (let's say a pie chart). You work on this during January and February, finally nailing it down at the start of March. Meanwhile, though, others in the project have spent February introducing a whole new way of accessing databases. By the time you make a pull request at the start of March things have changed a lot since you branched in January.

If you submit a PR without updating, the maintainers will likely ask you to update your branch to make it work with the new database system.

```
--January-|------February--------------|--March


          __pie_chart_branch___________
         /                             \
master--|--|-----------------------|----|-------
            \                      /
             \__new_database_____/
```

First thing to do is to update your local repository with the changes from upstream.

```
git pull upstream master
```

Then you could try two options:

1. Either merge `master` into `pie_chart_branch` yourself
2. Rebase `pie_chart_branch` on master (as though `pie_chart_branch` was created in late February and you did all the work very quickly!)

Option 1 is possible, but often merging your work involves touching parts of the system you don't know what much about and is better left to the core developers. In addition, merging in this way leaves merge commit messages and some projects really don't like those because they make the history harder to read.

Option 2 is generally preferred, since it focuses on clear communication via PRs that are easier to read and review.

Option 2 is called `rebase` and is explained usefully at this page from the EdX project. As you rebase, you can also `squash` some of your commits (treat many commits as one) to make them easier to follow for those reviewing your pull request. See the link above for details.

Removing something from history entirely.

The purpose of `git` is to retain all of your history, so that you can go back to any point in development and recover (as well as experiment while not breaking the mainline of development). Simultaneously when we are working in the open that means that anyone can view any file that was ever in a repository. With that in mind it is not too surprising that if you accidentally add something to git and then push it to github you can have trouble putting “the genie back in the bottle.”

Let’s say that we create a repo and add a README, then add a SPECIAL_SECRET file with the password “swordfish” in it. Note that I use `git add *` below which is a very common way to accidentally add a problematic file, try to get into the habit of adding files one by one.

```
SOI-A14570-Howison:PeerProduction howison$ cd practice_history_edit/

git init

Initialized empty Git repository in /Users/howison/Documents/UTexas/Co
urses/PeerProduction/practice_history_edit/.git/

vi README

git status

On branch master


No commits yet


Untracked files:

  (use "git add <file>..." to include in what will be committed)


        README


nothing added to commit but untracked files present (use "git add" to
track)

git add README

git commit -m "now we have a README"

[master (root-commit) f4878b0] now we have a README

 1 file changed, 1 insertion(+)

 create mode 100644 README

vi SPECIAL_SECRET

git add *

git commit -m "whoops added secret"

[master 018f6b5] whoops added secret

 1 file changed, 1 insertion(+)
```

```
 create mode 100644 SPECIAL_SECRET

git log --oneline --abbrev-commit --all --graph --decorate --color

* 018f6b5 (HEAD -> master) whoops added secret

* f4878b0 now we have a README
```

Now I'll go ahead and make one more edit to README

```
vi README

git add README git commit -m "README edit 2"[master 4d51f91] README edi
t 2

 1 file changed, 1 insertion(+)

git log --oneline --abbrev-commit --all --graph --decorate --color* 4d
51f91 (HEAD -> master) README edit 2

* 018f6b5 whoops added secret

* f4878b0 now we have a README

ls

README         SPECIAL_SECRET

cat SPECIAL_SECRET

swordfish
```

Ok, so we realize that the password file got into git and we swing into action and delete it from git.

```
git rm SPECIAL_SECRET

rm 'SPECIAL_SECRET'

git commit -m "phew removed it, or did we"

[master ff229ba] phew removed it, or did we

 1 file changed, 1 deletion(-)

 delete mode 100644 SPECIAL_SECRET

ls

README
```

So now the file is not there. Or rather it is not in our working directory. The problem is that it is still inside out `.git` folder and we can get it out easily.

```
git checkout HEAD~1

Note: checking out 'HEAD~1'.


You are in 'detached HEAD' state. You can look around, make experiment
al

changes and commit them, and you can discard any commits you make in t
his

state without impacting any branches by performing another checkout.
```

```
If you want to create a new branch to retain commits you create, you m
ay

do so (now or later) by using -b with the checkout command again. Exam
ple:


  git checkout -b <new-branch-name>


HEAD is now at 4d51f91... README edit 2

lsREADME          SPECIAL_SECRET

cat SPECIAL_SECRET

swordfish
```

Here I just used `git checkout HEAD~1` which goes one commit back in time, to before we deleted the SPECIAL_SECRET file. Even if we were far ahead, or over on other branches etc, I could always get back by asking to see the code just after the commit that added the file `git checkout 018f6b5` (btw, to get out of `DETACHED HEAD` state just checkout the branch again, we're working on master so it would be `git checkout master`).

So using `git rm` removes a file from the working directories but it doesn't remove it from the git history. And that's a sensible thing, usually you want to be able to go back in time. But sometimes you want to remove something from the history entirely. You can do that using the approaches outlined by Github here: Removing sensitive data from a repository

The process is a bit complex (as it should be) but simplified with the `bfg` tool, as described at the link above. First you have to download the tool (which requires Java to run) then follow the instructions step by step.

Keep in mind that if you had pushed this sensitive info to a repo on github and others had then forked or cloned it then that info is not going to be deleted from the clones, so passwords should definitely be changed and you should ask everyone to delete forks/clones and start again.

THere are a set of approachs to avoid uploading sensitive data. A good starting point is discipline around using `.gitignore` which will prevent adding files that should not be added. Another approach is to become familar with using environment variables to hold secrets. This is an evolving area, so ask others how they handle secrets (usually access credentials) when using git.