# Fracz git exercises

**Note:** git tree is not a git command, it is an alias I defined with:   
git config –global alias.tree “log –graph –decorate –all –oneline”   
it helps me visualizing the log in a clear way.

## 5. Chase-branch

We are on chase-branch branch. And there is also another branch, escaped, that has two more commits.

A white background with black text

Description automatically generated

You want to make chase-branch to point to the same commit as the escaped branch.

A diagram of a code

Description automatically generated with medium confidence

**Solution:**

As we can see in the following screenshot, the objective branch is two commits ahead of the branch that we are currently working on:

A black screen with white text

Description automatically generated

A diagram of a network

Description automatically generated

To bring the commits to the chase-branch, we need to merge escaped to our current branch:

A computer screen with text

Description automatically generated

A diagram of a diagram

Description automatically generated

## 6. Merge-conflict

Merge conflict appears when you change the same part of the same file differently in the two branches you're merging together. Conflicts require developer to solve them by hand.

Your repository looks like this:

A computer screen shot of a diagram

Description automatically generated

You want to merge the another-piece-of-work into your current branch. This will cause a merge conflict which you have to resolve. Your repository should look like this:

A diagram of a diagram

Description automatically generated

**Solution:**

Same way as before, we have two branches:

A screenshot of a computer program

Description automatically generated

Merging another-piece-of-work to our current branch results in a conflict that we must handle manually:

A screenshot of a computer

Description automatically generated

We just simply need to decide which one is the final result we want to keep and save that final version:

A screenshot of a computer

Description automatically generated

Once we are happy with the result, all we have to do is stage the file and commit it:

A computer screen with text

Description automatically generated

## 7. Save your work

You are working hard on a regular issue while your boss comes in and wants you to fix a bug. State of your current working area is a total mess so you don't feel comfortable with making a commit now. However, you need to fix the found bug ASAP.

Git lets you to save your work on a side and continue it later. Find appropriate Git tool and use it to handle the situation appropriately.

Look for a bug to remove in bug.txt.

After you commit the bugfix, get back to your work. Finish it by adding a new line to bug.txt with:

Finally, finished it!

Then, commit your work after bugfix.

**Solution:**

If we check the status of the current branch, we can see we have changes in some of the files that have not been committed:

A screen shot of a computer

Description automatically generated

Since those files are not finished, they can’t be committed yet, but neither can we switch to another branch to fix the issue. This is the current status:

A black screen with text and symbols

Description automatically generated with medium confidence

The solution here is to save those changes in a separate area with the stash command, and we can verify that by running git status to see that now, there are no changes to commit:

A computer screen shot of text

Description automatically generated

What we have done here is something like this:

A diagram of a computer program

Description automatically generated

We have temporarily saved those files with the changes we were working on in a separate area, so now we can proceed with other things, like fixing the bug:

A screenshot of a computer program

Description automatically generated

I´ll remove that line and commit the changes in the file:

A computer screen shot of text

Description automatically generated

Once the bug is fixed, we can recover our previous work with git stash pop:

A computer screen with white text

Description automatically generated

We can check the history of commits now:

A screen shot of a computer screen

Description automatically generated

With stash and stash pop, we have been able to “hide” our changes, work on other things, and bring them back later to continue our work:

A diagram of a computer program

Description automatically generated

Now I´ll just add the last line in the bug.txt file as stated in the problem:

A screenshot of a computer

Description automatically generated

Save and commit the file:

A computer screen shot of text

Description automatically generated

## 8. Change branch history

You were working on a regular issue while your boss came in and told you to fix recent bug in an application. Because your work on the issue hasn't been done yet, you decided to go back where you started and do a bug fix there.

Your repository look like this:

A computer screen shot of a text

Description automatically generated with medium confidence

Now you realized that the bug is really annoying and you don't want to continue your work without the fix you have made. You wish your repository looked like you started after fixing a bug.

A computer code with text

Description automatically generated with medium confidence

Achieve that.

**Solution:**

Let me rephrase a bit the issue here, the problem is that we were working in our branch, change-branch-history, we were asked to fix a bug in another branch, what we did, and then, came back to our original branch. We then realized that we want to incorporate the bug fix in our branch instead of having to deal with it again.

The current status of the history of commits is the following:

A screen shot of a computer

Description automatically generated

Before continuing our work, we would like to have the change introduced in the hot-bugfix branch in our current branch.

Merging the two branches would output the following result:

A computer screen shot of text

Description automatically generated

A diagram of a data flow

Description automatically generated

We are incorporating the changes of the hot-bugfix in our branch, but instead of being able to use them to continue our work, they were added at the end. While the desired result would look more like the following schema:

A diagram of a computer code

Description automatically generated with medium confidence

Where we have added the changes of the hot-bugfix to our branch before continuing with the work we were doing.

Here is where the **rebase** (see side note about rebase) command comes handy, since it allow us to do exactly what we are looking for here. Before continuing I´m going to reset my branch to the commit before doing the merge with reset –hard <number of the commit>:

A computer screen shot of text

Description automatically generated

Simply writing git rebase <name\_of\_the\_branch\_to\_bring\_to\_current>, allows us to get the desired result:

A screen shot of a computer program

Description automatically generated

## 9. Remove-ignored

File **ignored.txt** is ignored by rule in **.gitignore** but is tracked because it had been added before the ignoring rule was introduced.

Remove it so changes in ignored.txt file are not tracked anymore.

**Solution:**

We have a file in our repository that is been tracked but it shouldn´t. We can verify that by changing the file and see if git detects any changes:

A screenshot of a computer

Description automatically generated

Git diff effectively detects that a new line has been added:

A screen shot of a computer

Description automatically generated

If we want to preserve the file but avoid git from tracking it, we can run:  
git rm --cached ignored.txt

Git will act as if the file had been deleted, but we still have it in the repository:

A screen shot of a computer program

Description automatically generated

We just need to commit the changes and continue.

A computer screen shot of white text

Description automatically generated

## 10. Case sensitive filename

You have committed a File.txt but then you realized the filename should be all lowercase: file.txt. Change the filename.

This one is tricky on Windows, or in any filesystem that treats File.txt and file.txt as the same files.

**Solution:**

We simply need to change the file name using:

git mv <previous\_name> <new\_name>

A screen shot of a computer program

Description automatically generated

Stage the changes and commit:

A screen shot of a computer program

Description automatically generated

## 11. Fix typo

You have committed file.txt but you realized you made a typo - you wrote wordl instead of world.

Edit previous commit so no one would realize you haven't checked the file before committing it.

Pay attention to the commit message, too!

**Solution:**

For this exercise the command that allow us to modify the previous commit without introducing a new one is **--amend** (the following info comes directly from gitexercises.fracz.com)

* When you want to change the last commit (the one that is pointed by HEAD), use

`git commit --amend`

* If you want to change only commited files but no edit message, use

`git commit --amend --no-edit`

* Moreover, you can skip git add command and update last commit with all current changes in working area:

`git commit --amend --no-edit -a`

All we need to do is to fix the typo in the file:

A screen shot of a computer

Description automatically generated

And commit our change with git commit -a --amend (don’t forget to change also the commit message):

A computer screen shot of a black screen

Description automatically generated

## 12. Forge date

You should have finished your work a week ago. However, you had some more important things to do so you have committed the work just now.

As a git expert, change the date of the last commit. Don't be modest - make it look like it was committed in 1987!

**Solution:**

As we can see in the documentation, commit can work with a wide range of parameters:

A screenshot of a computer program

Description automatically generated

In the screenshot, I have highlighted the two that are relevant to our current problem. As in the previous exercise, amend will let us change our last commit and, using date, we can also change the date when our commit was made.

As we can see from the log:

A computer screen shot of a computer code

Description automatically generated

The last commit reflects that the work was done recently (today is the 1st of August 2024).

Let´s run the command and see the changes:

A computer screen with white text and numbers

Description automatically generated

Now it´s like the work was done some years ago.

## 13. Fix old typo

While you were working you noticed a typographic error in file.txt - you wrote wordl instead of world.

Unfortunately, you have made another commit on top of the typo so simple git commit --amend is not enough.

Fix the typographic error by amending commit in history. Pay attention to the commit message, too!

**Solution:**

This is the actual status of the commit history:

A black screen with colorful text

Description automatically generated

A diagram of a diagram

Description automatically generated

The above image reflects the situation, we are currently in the commit 8b7ef2a, but the fix we need to do is in the previous one, so amend will not serve this time.

For this situations, git rebase offers a very powerful parameter (-i or --interactive), that lets us revisit previous commits and fix them sequentially. It is like if we were amending different previous commits in our history. Let´s see how we can make use of it, I´ll use the following command:

git rebase -i HEAD~2

Where HEAD~2 means that I want to take my HEAD pointer two commits before where it currently is, and by doing so, I´ll be able to amend/edit the last two commits. Let´s see it in action:

A black screen with white text

Description automatically generated

Right after writing the command in git, I´m prompted with the rebase message:

A screenshot of a computer program

Description automatically generated

Since I specified that I wanted to revisit the last two commits (with HEAD~2), git outputs those two, and offers a series of available options to revisit them. Since the objective of this exercise is to amend the typo, I´ll pick “edit” from the list of available option on the commit I want to modify (leave the rest as they are; deleting a line will result in deleting the commit):

A blue and white screen

Description automatically generated

Save and close the file. If I now execute git tree to see the history of commits:

A black screen with white text

Description automatically generated

We can see that our HEAD pointer is now in the commit we want to modify. What we need to do now is to open the file that contains the typo, fix it and commit the change with git commit -a --amend (or a sequence of git add file.txt and git commit --amend)

A screen shot of a computer

Description automatically generated

A black screen with white text

Description automatically generated

After doing so, we are prompted with the message that we wrote in the commit. I´ll change it to the correct message without the typo.

A screen shot of a computer

Description automatically generated

After saving and closing the file, git reminds us that we are still in the middle of a rebase:



To finish it, we will type git rebase --continue:

A screen shot of a computer

Description automatically generated

But git found a conflict in the file, we would need to manually fix it to continue our work:

A red arrow pointing to the right

Description automatically generated

Once fix, we need to commit it and we can finish the rebase:

A screen shot of a computer screen

Description automatically generated

## 14. Commit lost

You have created a commit with very important piece of work. You then wanted to fix something in the last commit so you have amended it. However, you have just realized you have accidentally committed the wrong changes and you desperately need the first version of the commit you have just amended.

However, there is no previous version in the history - you have edited the last commit with git commit --amend.

Your goal is to find the first version of the commit in the repository. It must be somewhere...

Once found, force the commit-lost branch to point at it again and verify the solution.

**Solution:**

We found ourselves in a situation where we have overwritten the last commit and then realized that we need to get the first version back. This is the current situation:

A black screen with colorful text

Description automatically generated

In this case, we have amended the last commit, but we could evenly be in the situation where we force-deleted a branch or hard-reset it and lose those changes from the history. For those cases, the question that arises is, is there a way to get these commits back?

Fortunately, there is. The command now is git reflog. As it states in the documentation:

“As you’re working, Git silently records what your HEAD is every time you change it. Each time you commit or change branches, the reflog is updated. (…) You can see where you’ve been at any time by running git reflog:” (<https://git-scm.com/book/en/v2/Git-Internals-Maintenance-and-Data-Recovery>)

A screen shot of a computer

Description automatically generated

What we can extract from this output is:

* The current state of HEAD (HEAD@{0}) is pointing to the current branch (commit-lost), and that the most recent commit here involved an amend operation.
* Previous state of HEAD (HEAD@{1}), had a comment indicating that it was a very important piece of work, and this is the point where we want to return.

To go back to the state our branch was at the previous state of the HEAD, we need to run the reset command, specifying the state of the HEAD we want to recover. After doing so, we can verify that we are where we wanted to be:

A computer screen with text

Description automatically generated

## 15. Split commit

You have committed both first.txt and second.txt as one commit. However, you made a mistake. You intended to commit first.txt in the first commit and the second.txt in the second one.

**Solution:**

Again, here we need to reset our files to a previous state. This is the current status:

A black screen with white text

Description automatically generated

To get back to the previous commit, we have different options with the same reset command:

* git reset 8d60c81 – Specifying the SHA of the commit
* git reset HEAD@{1} – As before, this will return the HEAD pointer to the previous state
* git reset HEAD^ – Same result but different notation. Adding more ^ will be the same as incrementing the number between curly brackets in the previous option:
  + reset HEAD^^ = reset HEAD@{2}

Another note about git reset, reset takes one parameter (mode) that can take three different possible values:

* soft: returns to the commit or state specified but maintains the files indexed in the stage area.
* mixed: returns to the commit or state specified but your files will be un-staged.
* hard: returns to the commit or state specified and removes all changes you made in the files committed.

(I strongly encourage you to play and test the different results of these three)

When we don’t specify any mode, git takes the mixed mode as default.

Here, I used the first option:

A screen shot of a computer

Description automatically generated

As we can see, the result of the mixed reset is that I recovered my files with the changes I made but those are not staged. At this point, what I need to do is to stage only the file I´m interested in and leave the other one out of this commit.

A computer screen shot of a black screen

Description automatically generated

Then add the second file and commit again:

A computer screen shot of text

Description automatically generated

## 16. Too many commits

You were working on an issue and created two commits introducing very small change. You don't want to mess up your project history so you want to make only one commit that contains changes made in the last two.

Execute git log -2 to see last two commits.

**Solution:**

Let´s do as the exercise states and run git log -2 to see the last two commits:

A computer screen with numbers and text

Description automatically generated

In this case, it seems that we forgot to add one line in a file. Since it is only one line and in one file, we could take the HEAD to the state of two previous commits (git reset HEAD^^), add the file with the two lines on it and commit the changes maintaining the previous comment (“Add file.txt”):

A computer screen shot of a program

Description automatically generated

But this is very manual, and it only works since we are only dealing with one file, if we had to manage more files and more changes this would be a tedious process.

The best approach is to use rebase again and squash the two commits in only one:



In the message, change pick for “s” or “squash” in the commit we want to squash with the previous one:

A close up of a text

Description automatically generated

Finally continue the rebase and confirm the message that will be left as the result of the squash:

A screen shot of a computer

Description automatically generated

A red arrow pointing to a computer screen

Description automatically generated

And we´ll be good to go.

## 17. Executable

You have created a simple bash script in script.sh. However, when you check it out on Unix, it does not have required execute permissions so you can't launch it with ./script.sh without performing chmod +x script.sh beforehand.

Fix it by adding an executable bit for script.sh in Git history.

**Solution:**

As the exercise states, we have an executable file that needs to run automatically without introducing chmod +x script.sh beforehand.

Let´s open the file to see its content:

A screenshot of a computer

Description automatically generated

This is a simple program that will only print: “Git exercises” in the console. We can execute the file, since we are in our laptop we won’t have any issues (the problem states that the issue comes when trying to execute it in another server):

A black background with white text

Description automatically generated

As we can see from the output, the program executes without issues. We can check the permissions of the file with ls -l script.sh:

A black screen with white text and numbers

Description automatically generated

This output gives us the following information:

* **-:** Indicates this is a regular file. If it were a directory, this would be a d.
* **rwx (red):** The owner (Daniel) has read (r), write (w), and execute (x) permissions.
* **r-x (blue):** The group has read and execute permissions, but not write. The write permission is represented by -.
* **r-x (green):** Others have read and execute permissions, but not write.

The problem states that, in other servers, the file is not executable directly and needs chmod +x script.sh to be executed beforehand. What that line does is to give the owner permission to execute the file. From that, we can infer that what happens is that when we upload the file, it loses its permission to be executed from another server. If we, or another user try to see the permissions, they would find something like this:

A number on a black background

Description automatically generated

Where the executable permission (x) is not granted.

To correct this issue, we need to use git-update-index (https://git-scm.com/docs/git-update-index#Documentation/git-update-index.txt---chmod-x) along with one of its parameters –chmod to set or remove the "executable" flag on any tracked file:

git update-index –chmod=+x script.sh

A computer screen shot of a program

Description automatically generated

## 18. Commit part of work

You are working on an issue for a long time and you noticed you have done too much. You want your work to be committed in two separate commits instead of one.

Unfortunately, your changes involve only one file so it is impossible to git add different files separately.

Commit all new lines that contains "Task 1" phrase in the first commit and the rest of them in the second.

**Solution:**

Git add has another interesting parameter (-p or --patch) that allow us to interactively choose which patches we want to select for the commit. It continuously shows small portions of the changed files and asks what to do, allowing us to choose in each step, what parts of the code we want to stage and what parts will be ignored now.

Let´s see it in action:

A computer screen shot of a program

Description automatically generated

At the end, git asks if we want to stage this part. From the available options, “s” allow us to split the code in smaller chunks, so I´ll press “s”.

A black screen with green and blue text

Description automatically generated

Now we can go line by line. I´ll press “y” only for those line containing “task 1” in them.

A computer screen shot of a program

Description automatically generated

We can verify that what we have staged is only lines containing “task 1” with git diff –cached:

A computer screen shot of a program

Description automatically generated

Now it´s only a matter of making the two commits:

A computer screen shot of text

Description automatically generated

## 19. Pick your features

You have implemented three different features of a program in three different local topic branches.

A group of black text

Description automatically generated

You want to have all these features as single commits in pick-your-features branch.

A computer code with black text

Description automatically generated

**Solution:**

This is the current status:

A screen shot of a computer

Description automatically generated

A diagram of a diagram

Description automatically generated

And what we want to achieve is this:

A diagram of a diagram

Description automatically generated

If we opt for merging, this is what we would get instead:

A screenshot of a computer program

Description automatically generated

A diagram of a diagram

Description automatically generated

To achieve the desired result, we need a way to bring the last commits of other branches to the current one without merging. This is achieved by cherry picking a branch (see end note about cherry picking).

Since we want to end with the result Z (current position) --> A --> B --> C, I´ll start by cherry-picking feature-a:

A screen shot of a computer

Description automatically generated

Then feature-b and feature-c (we´ll need to fix the conflicts):

A screenshot of a computer program

Description automatically generated

A computer screen shot of a computer program

Description automatically generated

Save the work and commit the changes:

A computer screen shot of a black screen

Description automatically generated

## 20. Rebase complex

You were working on an issue-555 topic branch. You noticed a bug, which you fixed in rebase-complex topic branch. Then, you finished issue-555.

However, you need to have bug fixed in your-master branch without any work done on issue-555.

Situation is as follows:

A computer code with text

Description automatically generated with medium confidence

Only H and I commits should be rebased onto D.

Try to do this with a single git command.

Solution:

This is the status of the commits:

A screen shot of a computer program

Description automatically generated

And we want these two commits in our master branch:

A computer screen with text

Description automatically generated

This could be done by switching to your-master branch and cherry-picking the rebase-complex as we did in the previous exercise, but the problem states to try to do it with a single command.

Another way to state the problem is that we want to bring all changes into your-master but not those ones that belong to issue-555, since we are still working on those and that branch can´t be incorporated.

This can be achieved by using rebase --onto as the documentation states (source: <https://git-scm.com/docs/git-rebase>)

A screenshot of a computer

Description automatically generated

The syntax is the following:

git rebase --onto <new\_base> <upstream> <branch>

Where:

* <new\_base>: The commit or branch where you want to move your commits to.
* <upstream>: The commit or branch from where the rebase should start. All commits after this point in the specified branch will be moved (unless we specify a branch, in which case only the commits affecting that branch will be moved).
* <branch>: The branch you want to rebase.

Applying that to our case would result in the following:

git rebase --onto your-master issue-555 rebase-complex

* New\_base is your-master, since we want to bring the commits to this branch.
* Upstream is issue-555, since we want to bring all changes that started with that branch.
* Branch is rebase-complex, since we only want to bring the commits that started with issue-555 but that belong to rebase-complex.

A computer screen shot of a program

Description automatically generated

## 21. Invalid order

You have committed two changes but you don't like the order in which they appear in the history. Switch them.

To show commits that needs switching, execute git log -2 command.

**Solution:**

Let´s check the log first:

A computer screen with text and numbers

Description automatically generated

With the commands that we’ve seen so far this should be easy. The only thing we need is to do an interactive rebase, taking the HEAD pointer two commits before the current status, and change the order of the commits:

A black screen with white text

Description automatically generated

When prompted, simply switch the order of the commits:



Save and close the file.

A computer screen shot of a program

Description automatically generated

## 22. Find swearwords

You noticed that your project contains swearwords. You don't want to remove them in the next commit as your boss might someday find out that they were present in the codebase for a while.

Find all commits that add shit word to either words.txt or list.txt and change them so they introduce word flower instead.

**Solution:**

Here we face a similar issue that the one we had on problem 13 (fix old typo). At this point we know that we can use rebase in an interactive way to amend previous commits, so the new aspect here would be to find those commits that introduced the word shit in our work and change them for the word flower. With a simple search in google we can find the following information:

A close-up of a computer screen

Description automatically generated

Source of the screenshot: <https://stackoverflow.com/questions/5816134/how-to-find-the-git-commit-that-introduced-a-string-in-any-branch>

Let´s understand the commands:

* **-S <search\_string>:** This searches for changes that add or remove instances of the specified string in the codebase.
* **--source:** This option shows which commit was changed from which branch, helping to understand where the changes originated from (only for output formatting).
* **--all:** This searches through all branches in the repository, not just the currently checked-out branch.

This is the current state of our commit history:

A screen shot of a computer

Description automatically generated

We see there that we have only been working with a single branch, so the --all parameter here won´t be needed, since we are only interested in swearwords in our branch. On the other hand, --source is particularly useful in scenarios where you want to identify the branch origin of specific commits. However, since we were only working in a single branch and all commits are made exclusively there, using --source may not provide additional information because all changes are originating from the same branch.

So let´s see how it works:

A screenshot of a computer program

Description automatically generated

git log -S shit outputs all the commits where the word was introduced. Now that we have them, it will only be a matter of amending them with an interactive rebase.

As shown two screenshots above, we find ourselves in the commit 105 so I´ll use this number to see all the commits in the edit mode of the rebase but I´ll only edit those that appeared in the above image (23, 46, and 94):

A black background with green and white text

Description automatically generated

A screenshot of a computer

Description automatically generated

Once saved, we can start amending the commits. At this point, another command comes in handy: git log -p -1, since it allows us to see what was added in this commit:

A computer screen shot of a computer code

Description automatically generated

This means that in this commit, the word “shit” was added to list.txt. I´ll go there and change it.

A screenshot of a computer

Description automatically generated

Save and amend the changes. We can verify the changes before committing them:

A computer screen shot of a black screen

Description automatically generated

And then commit:

A screen shot of a computer program

Description automatically generated

Now we can continue with the rebase:

A screenshot of a computer screen

Description automatically generated

We only need to repeat the process and change all swearwords.

A computer screen shot of a black background

Description automatically generated

And one last time:

A computer screen shot of a black screen

Description automatically generated

Finally, we can finish the rebase:

A black background with white text

Description automatically generated

## 23. Find bug

Your customer claims that there is a bug in application. The word jackass is being displayed on main screen.

He can not tell when the word appeared the first time. However, he is sure that there was no jackass in the version 1.0 of the application. He want you to find who has added this and fix it ASAP.

However, the task is not so simple. It turns out that the home screen text is encoded in source code with base64 algorithm for sanity reasons. It is impossible to search for commit that introduces this swearword with git log -S command. What's more, the text in home screen has been changed in the last 300 commits.

Your task is to find the first commit that introduces the jackass word and push it for verification

**Justification**

Normally you don't face base64 encoded strings that you need to search in. However, this perfectly simulates common situation when *something were working back then but now is broken*. You often can't even tell where and when the bug could be introduced. In *real life* you would write an unit test that verifies if the bug exists. This would help you to find a commit introducing bug dramatically.

**Useful tips**

* First of all, you don't want to search for *jackass* commit by hand.
* You can find last known working (no *jackass*) version of the project by 1.0 tag.
* You can decode contents of the home page text with the following command

openssl enc -base64 -A -d < home-screen-text.txt

* grep can help you verify whether the decoded content contains *jackass* or not. It's worth to know the grep -v option that inverts default grep behavior - returns status code 0 if the word has not been found and 1 otherwise
* you can run any command in shell with sh -c "any command"
* Use information above to create a simple unit test that would help to automate searching for a first commit with bug
* When you find the first commit with *jackass*, you can push it for verification with the following command

git push origin COMMIT\_ID:find-bug

**Solution:**

Git has a very useful tool to find bugs in our branch and that is git bisect. This command is used to find where a bug or an issue was introduced into the code base. In simple terms, let´s picture the following scenario, we have the following succession of commits:

A diagram of two circles

Description automatically generated

git bisect allow us to declare a known good state and a known bad state:

A diagram of a circle

Description automatically generated with medium confidence

Then, we´ll be asked if a commit in between those states is good or bad; after we label it as good or bad, git will continue asking for this labelling for different commits in between the commit labelled as good and the last commit labelled as bad until we find the commit that introduced the bug.

A diagram of a diagram

Description automatically generated

A diagram of a good and bad

Description automatically generated

A diagram of a computer network

Description automatically generated

Another thing that is mentioned here is the **tag**. Tagging is a useful tip to identify specific commits by a given alias instead of the SHA number (the seven first characters of the commit identifier that allow us to differentiate between commits). The problem states that version tagged as 1.0 didn´t have the bug. Therefore, we´ll categorize the head as bad and this commit as good. We start the process with git bisect start, followed by the commands that identify the bad and good commits:

A computer screen with text

Description automatically generated

Git informs that we have around 7 steps to check and is asking now to label if commit 96304d1 is good or bad. We could check the content of the commit with git show 96304d1:

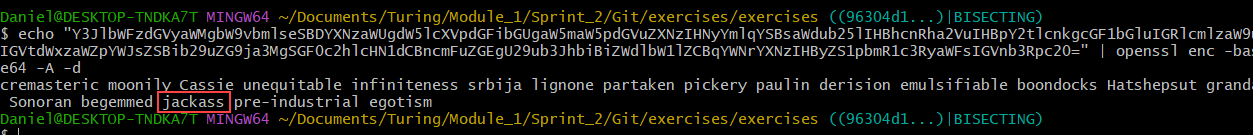
A screen shot of a computer program

Description automatically generated

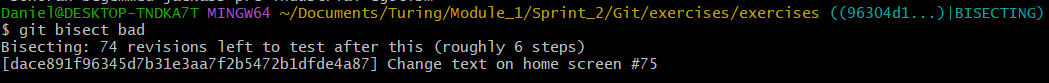
We can see that a line was remove and another one added, but as the problem states, the information is encoded so it´s not very user friendly as it is. There are two ways to solve the issue, one being the manual way and the other automated.

**Manual:**

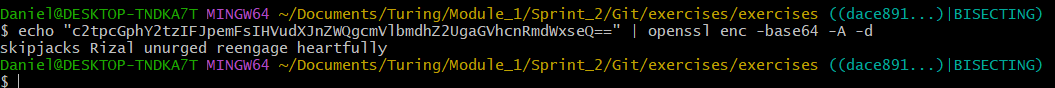
Since there are 7 steps to check, we can go one by one, as the process I explained before. In order to do so, I take the encoded message and decode it in the console with echo “message\_to\_decode” | openssl enc -base64 -A -d (as the instruction in the problem states):



We can see the word jackass there. Then, this commit should be labelled as bad:



Now the commit to check is dace891:

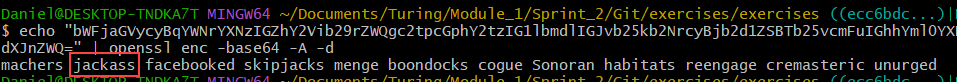


No trace of “jackass” here, so we can label it as good and continue with the next one:

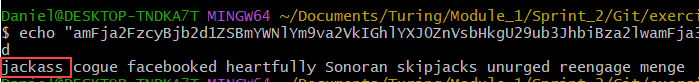
A computer screen shot of text

Description automatically generated

Bad.



Bad



Bad

A screenshot of a computer screen

Description automatically generated

Bad

A black background with white text

Description automatically generated

Good. And last one:

A computer screen shot of text

Description automatically generated

Bad

Now that we have found the first commit that introduced the bug, we can use git bisect reset to go back to the previous state of the HEAD:

A computer screen shot of text

Description automatically generated

Now we should go back to the commit where we introduced the change #78 and fix the issue.

This is the manual way, and it could be done since there are only 7 steps to check but there´s another, more elegant and fast way to do so.

**Automated:**

Instead of manually label every step, we can automate the search for the first appearance of the word jackass with the following command:

git bisect run sh -c "openssl enc -base64 -A -d < home-screen-text.txt | grep -v jackass"

Let´s explain its functionality:

**git bisect run:**

* This command automates the git bisect process by running a specified command at each step of the bisect process.
* It expects a command that will exit with status 0 (indicating success or "good") or status 1 (indicating failure or "bad").
* It continues the bisect until it finds the first commit that causes the command to exit with a non-zero status.

**sh -c "...":**

* This invokes a new shell instance to execute the command string provided within quotes.
* It allows you to run complex command sequences that involve pipes or redirection.

**openssl enc -base64 -A -d < home-screen-text.txt:**

* openssl enc -base64 -A -d: This part of the command uses OpenSSL to decode the content of a file from Base64 encoding.
  + -base64: Specifies the Base64 encoding/decoding mode.
  + -A: Processes the input as a single line, ignoring newlines, which is useful for Base64-encoded data.
  + -d: Tells OpenSSL to decode the input.
* < home-screen-text.txt: Redirects the content of the file home-screen-text.txt to OpenSSL for decoding.

**| grep -v jackass:**

* |: Pipes the decoded output from the OpenSSL command to grep.
* grep -v jackass: Searches the decoded content, excluding lines that contain the word "jackass".
  + -v: Inverts the match, meaning it only prints lines that do not contain the specified string.

The goal of the command checks whether the Base64 decoded content of the file home-screen-text.txt contains any lines with the word "jackass".

Output Behavior:

* Exits with Status 0 (good): If no lines containing "jackass" are found after decoding, the command exits successfully.
* Exits with Status 1 (bad): If any line containing "jackass" is found, the command exits with a non-zero status.

During the git bisect process, git bisect run uses the exit status of this command to determine whether a commit is "good" or "bad":

* Good Commit: The command exits with status 0, indicating the commit is not the source of the unwanted text.
* Bad Commit: The command exits with status 1, indicating the commit introduces or retains the unwanted text.

The start here would be exactly the same as before:

A computer screen with white text

Description automatically generated

But then, instead of going one commit at a time, we can use the above command to automate the search:

A computer screen shot of a black screen

Description automatically generated

Again, we end up with the commit #78 identified as the first bad commit.

To finish, we can do a git bisect reset and push the first commit that introduced the error to the origin:

A screenshot of a computer program

Description automatically generated

A computer screen shot of a program

Description automatically generated

## Rebase

To understand the difference between merging and rebasing I’m going to create a new repository where I’ll be working with different files and comparing the result of using both techniques.

First, I initialize an empty repository in a new folder I called tests:

A screenshot of a computer

Description automatically generated

Then, I change the name of my branch from master to main and create a new text file:

A screen shot of a computer program

Description automatically generated

Before creating a new branch, I need to do an initial commit:

A screen shot of a computer

Description automatically generated

Now, I will create a new branch to do separate work and I’ll switch to that branch:

A screenshot of a computer screen

Description automatically generated

Here, I’m adding a new file called “shopping\_list.txt” where I’ll include some things I want to buy.

A screen shot of a computer

Description automatically generated

I’ll save my changes and commit them:

A screen shot of a computer

Description automatically generatedNow I’ll switch to the main branch and work a little there.

A screenshot of a computer

Description automatically generated

I added some lines in the list.txt file and commit them. After the commit, this is how the log looks like:

A screenshot of a computer screen

Description automatically generated

The two branches have experienced changes in their states, and here is when the difference arises. If now I would like to merge the two branches, I will end up with something like this:

A screen shot of a computer program

Description automatically generated

The file has been added to the main branch as intended, but if we take a closer look at the history, it shows that the work was done separately in the branches.

Now, to be able to compare the results, I’ll do a hard reset to return to the state before the merge:

A screen shot of a computer program

Description automatically generated

After the reset, there’s no trace of the shopping\_list file in the main branch, and the repo shows the same history as before the merge.

If I use git rebase instead, the result is the following:

A screen shot of a computer program

Description automatically generated

It added the file to the main branch as expected, but it also merged the history of the commits as if they had happened in a single branch. The merge here was simple, since there was only one commit, and it was done before the commit on the main branch (adding my morning routine). To see more functionality, I´ll revert the changes again and I’ll do another commit in the shopping list:

* Hard reset again

A screenshot of a computer screen

Description automatically generated

* Add a few more lines to the shopping list:

A screenshot of a computer

Description automatically generated

* Commit the changes:

A screenshot of a computer

Description automatically generated

Now the log looks like this:

A screenshot of a computer

Description automatically generated

If I now switch to the main branch and rebase shopping\_list, this is what I get:

A screen shot of a computer program

Description automatically generated

Rebase has not only brought all the changes to the main branch, but it also ordered them, so they come before the last commit in main.

Let’s compare the result now if I had instead run merge:

A computer screen shot of text

Description automatically generated

I first run a git reset –hard to go back to the point where I did not have the two branches together, and then, I run merge instead. The result is different as we can see from the below two screenshots:

A screenshot of a computer

Description automatically generated

Figure 1. Result of rebasing

A screenshot of a computer screen

Description automatically generated

Figure 2. Result of merging

## Cherry picking

I’ll explain Cherry pick now. I have rolled back the last changes and go back to the status where the two branches were separated.

A screenshot of a computer screen

Description automatically generated

I am currently in the main branch, and maybe I find myself in a situation where I would like to include the first commit in the shopping\_list (a913694) branch into my main branch, but not the last one. I could do that with the following command:

git cherry-pick a913694

A computer screen shot of a program

Description automatically generated

This allows us to bring commits from different branches to the current branch without pulling all the changes.