1. (5 Punkte) In einem Repository steht in der Datei `f.txt` nur der Buchstabe A. Arbeitsverzeichnis, Staging Area und letzter Commit stimmen überein. Gib nach jedem Schritt an, was in `f.txt` im Arbeitsverzeichnis, in der Staging Area und im letzten Commit steht.

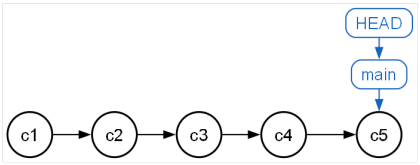
	Schritt	Arbeitsverzeichnis	Staging Area	letzter Commit
	Anfang	A	A	A
1.	Datei auf B ändern	B	A	A
2.	`git add f.txt`	B	B	A
3.	Datei auf C ändern	C	B	A
4.	`git commit -m x`	C	B	B
5.	`git restore f.txt`	B	B	B

Lösung: `git commit` sichert den Stand der Staging Area (B), nicht den des Arbeitsverzeichnisses (C). `git restore` holt die Datei aus der Staging Area zurück, die Änderung auf C ist verloren.

2. (4 Punkte) Gib jeweils einen Git-Befehl an.
   (a) Du hast `programm.py` geändert und willst die Änderung verwerfen (noch nicht mit `add` gestaged).
   (b) Du hast `programm.py` versehentlich mit `git add` in die Staging Area gelegt und willst sie dort wieder herausnehmen, ohne die Änderung zu verlieren.
   (c) Du willst kurz auf einen anderen Branch wechseln, hast aber noch ungesicherte Änderungen, die du später weiterbearbeiten willst.
   (d) Du willst einen neuen Branch `grafik` anlegen und gleich dorthin wechseln.
   (e) Du willst sehen, was du seit dem letzten Commit geändert hast.
   (f) Du willst sehen, wie die Datei `programm.py` vor zwei Commits aussah, ohne etwas zu verändern.

Lösung:
(a) `git restore programm.py`
(b) `git restore --staged programm.py`
(c) `git stash`, danach wechseln, zurückwechseln und `git stash pop`
(d) `git switch -c grafik`
(e) `git diff` (Arbeitsverzeichnis gegen Staging Area) oder `git diff HEAD` (gegen den letzten Commit)
(f) `git show HEAD~2:programm.py` oder `git switch --detach HEAD~2` (danach `git switch main`)

3. (4 Punkte) Gegeben ist die folgende Geschichte eines Repositorys.



(a) Welche Commits sind `HEAD~2` und `HEAD~4`?

(b) Nach `git switch --detach HEAD~3`: Auf welchen Commit zeigt HEAD? Was bedeutet „detached“?

(c) Wie kommst du danach wieder auf den neuesten Stand?

Lösung:

(a) `HEAD~2` ist c3, `HEAD~4` ist c1.

(b) Auf c2. HEAD zeigt direkt auf einen Commit statt auf einen Branch. Man kann sich den Stand ansehen; neue Commits gehörten zu keinem Branch.

(c) `git switch main` (oder `git switch -`).

4. (5 Punkte) In einem neuen Repository werden nacheinander diese Befehle ausgeführt. Bei jedem Commit wurde vorher eine Datei geändert, die Commit-Nachricht nennen wir als Namen des Commits.

```
git commit -m c1   git commit -m c2   git switch -c neu   git commit -m n1
  git commit -m n2   git switch main   git commit -m c3   git merge neu
```

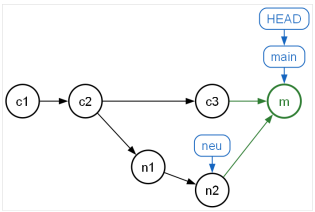
(a) Zeichne den Commit-Graphen mit den Branches und HEAD.

(b) Welche Art von Merge findet statt? Warum?

(c) Wie viele Vorgänger hat der letzte Commit?

Lösung:

(a) m ist der Merge-Commit.



(b) Ein Merge mit Merge-Commit (kein Fast-Forward): Nach dem Abzweigen von `neu` kam auf `main` noch c3 dazu, die Branches haben sich auseinanderentwickelt.

(c) Zwei: c3 und n2.

5. (4 Punkte) Auf `main` wird `git merge design` ausgeführt. Git meldet einen Konflikt, die Datei `farben.txt` sieht danach so aus:

```
rot
<<<<<<< HEAD
weiß
=======
gelb
>>>>>>> design
blau
```

   (a) Wie kam es zu dem Konflikt? Was stand in der Datei, bevor sich die Branches getrennt haben (soweit man es erkennen kann)?

   (b) Welche Fassung stammt aus `main`, welche aus `design`?

   (c) Der Konflikt soll so gelöst werden, dass beide Farben in der Datei stehen. Gib die Datei und die Befehle an, mit denen der Merge abgeschlossen wird.

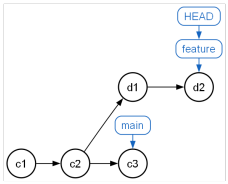
Lösung:

(a) Beide Branches haben dieselbe Zeile (die zweite) verschieden geändert, Git kann nicht entscheiden, welche Fassung gilt. Die erste und letzte Zeile (rot, blau) waren in beiden gleich; was vorher in der zweiten Zeile stand, sieht man hier nicht.

(b) Zwischen `<<<<<<< HEAD` und `=======` steht die Fassung von `main` (weiß), darunter bis `>>>>>>> design` die von `design` (gelb).

(c) Datei: rot, weiß, gelb, blau (je eine Zeile, ohne die Markierungen). Dann `git add farben.txt` und `git commit` (oder `git commit -m „merge design“`).

6. (5 Punkte) Gegeben ist die folgende Situation, HEAD steht auf `feature`.



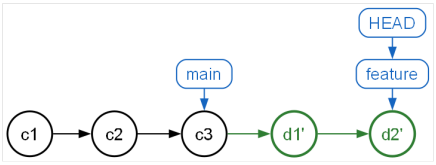
(a) Zeichne den Graphen nach `git rebase main`.

(b) Danach werden `git switch main` und `git merge feature` ausgeführt. Welche Art von Merge ist das? Zeichne das Ergebnis.

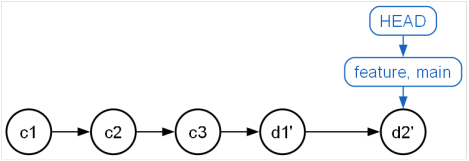
(c) Warum soll man `rebase` nicht auf Commits anwenden, die andere schon haben?

Lösung:

(a) d1 und d2 werden als neue Commits d1', d2' hinter c3 gesetzt:

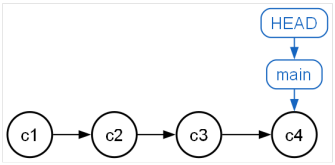


(b) Ein Fast-Forward-Merge, `main` wird nur nach vorne geschoben:



(c) d1' und d2' sind neue Commits mit anderen Hashes. Wer d1 und d2 schon hat, dessen Geschichte passt nicht mehr zu unserer; beim nächsten Abgleich entstehen doppelte Commits und Konflikte.

7. (5 Punkte) Der Commit c4 enthält einen Fehler und soll rückgängig gemacht werden.



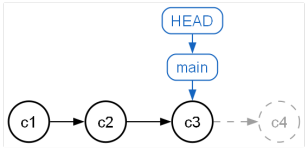
(a) Zeichne den Graphen nach `git reset --hard HEAD~1` und (ausgehend von der Abbildung) nach `git revert HEAD`.

(b) Welchen Befehl nimmst du, wenn c4 schon auf GitHub liegt und andere damit arbeiten? Begründe.

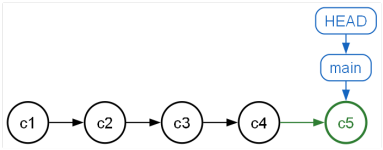
(c) Was ist bei `git reset HEAD~1` (ohne `--hard`) anders?

Lösung:

(a) reset: `main` zeigt auf c3, c4 gehört nicht mehr zum Branch.



revert: Ein neuer Commit c5 nimmt die Änderungen von c4 zurück.



(b) `git revert`: Die Geschichte bleibt erhalten, es kommt nur ein Commit dazu. Nach einem reset würde c4 bei uns fehlen, bei den anderen aber nicht, die Geschichten passen nicht mehr zusammen.

(c) Die Änderungen von c4 bleiben im Arbeitsverzeichnis erhalten (nicht gestaged), nur der Commit ist weg. Bei `--hard` werden auch sie verworfen.

8. (4 Punkte) Die Datei `.gitignore` eines Repositorys enthält:

```
*.log
geheim.txt
notizen/
*.tmp   # temporäre Dateien
```

(a) Welche der folgenden Dateien werden von Git ignoriert?

`fehler.log`, `logs/alt.log`, `geheim.txt`, `privat/geheim.txt`, `notizen/idee.txt`, `test.tmp`, `programm.py`

(b) `programm.py` ist schon in einem Commit. Jetzt wird `programm.py` in `.gitignore` eingetragen. Was passiert bei der nächsten Änderung an der Datei? Was muss man tun?

Lösung:

(a) Ignoriert: `fehler.log`, `logs/alt.log`, `geheim.txt`, `privat/geheim.txt`, `notizen/idee.txt`.
Nicht ignoriert: `test.tmp` und `programm.py`. In der letzten Zeile gehört der Kommentar zum Muster, weil # nur am Zeilenanfang einen Kommentar einleitet. Das Muster passt deshalb auf keine Datei.

(b) Git meldet die Änderung trotzdem, `.gitignore` wirkt nur auf Dateien, die noch nicht verwaltet werden. Man muss die Datei mit `git rm --cached programm.py` aus der Verwaltung nehmen und das committen.