



Revisjonshistorie

År	Forfatter
2020	Kolbjørn Austreng
2021	Kiet Tuan Hoang
2022	Kiet Tuan Hoang
2023	Kiet Tuan Hoang
2024	Terje Haugland Jacobsson Tord Natlandsmyr

I Introduksjon - Praktisk rundt filene

I denne øvingen får dere ikke utlevert noen `.c` eller `.h`-filer.

II Introduksjon - Praktisk rundt øvingen

Versjonskontroll er en måte å ta *bilder* av en fil på, slik at man kan se hvordan filen har endret seg, og når endringene ble gjort. Programmet `git` er et utbredt verktøy for versjonskontroll. Versjonskontroll er spesielt viktig når man jobber flere i lag på de samme filene. `git` ble opprinnelig utviklet av Linus Torvalds i 2005 for å gjøre det lettere for flere programvareutviklere å samarbeide på Linux-kjernen samtidig. Til og med `git`-prosjektet bruker `git` for versjonskontroll.

Denne øvingen er ment som en introduksjon til praktisk bruk av `git`. For å illustrere de mest brukte delene av `git` vil øvingen være strukturert som en walk-through og består av å skrive enkel C-kode, som skal versjonskontrolleres (introduksjon [III](#)). Oppgaven finner dere i seksjon [1](#) og handler om at dere skal vise at dere mestrer `git` til en studass for å få godkjent øvingen.

II .1 Avhengigheter

Denne øvingen krever at man har `git` på datamaskinen før man begynner. Datamaskinene på Sanntidssalen har `git` installert, men det kan også være greit å skaffe

det selv også, ettersom det er et nyttig verktøy. Dette kan gjøres via <https://git-scm.com>, eller via en pakkebehandler om operativsystemet har en¹.

I tillegg til `git`, trenger man en C-kompilator og tekstbehandler for å kompilere og skrive C-kodene vi kommer til å versjonskontrollere. Denne oppgaveteksten kommer til å bruke `gcc`, men en hvilken som helst C-kompilator vil fungere (datamaskinene på Sanntidssalen har allerede `gcc` installert).

III Introduksjon - Grunnleggende Kommandoer

III .1 Oppsett

Åpne en terminal og skriv inn `git --version`. Dette er en enkel test for å sjekke om `git` er riktig installert og ligger i `PATH`-variabelen til operativsystemet.

Før `git` kan brukes må man gjøre noen enkle konfigurasjoner. Først og fremst må `git` ha litt informasjon om brukeren. Dette må bare gjøres en gang, og består av to kommandoer:

```
student@Ubuntu:~$ git config --global user.name "Linus Torvalds"
student@Ubuntu:~$ git config --global user.email "linux@stud.ntnu.no"
```

Flagget `--global` fører til at `git` lagrer navn og email i filen `~/.gitconfig`. Om man jobber på en datamaskin andre bruker - slik som i heisprosjektet, er det lurt å konfigurere `git` lokalt ² (**VIKTIG!**). Det gjøres fra et allerede opprettet `git`-repository, altså at kommandoen `git init` har blitt brukt i mappen, ved å kalle de samme kommandoene uten `--global`-flagget inne i mappen til `git`-repositoryet:

```
student@Ubuntu:~$ git config user.name "Linus Torvalds"
student@Ubuntu:~$ git config user.email "linux@stud.ntnu.no"
```

For ekstra brukervennlighet, har `git` muligheten til å definere aliaser for lange kommandoer som ofte brukes. For eksempel brukes kommandoen `git checkout` ofte slik at mange liker å aliase denne til `git co`. For denne øvingen definerer man aliaset `lg` (betyr det samme som `git log --all --oneline --graph --decorate`):

```
student@Ubuntu:~$ git config --global alias.lg "log --all --oneline
--graph --decorate"
```

Akkurat dette aliaset er spesielt nyttig for videre bruk av `git`.

¹apt, yum, pacman etc for Linux. Homebrew for mac

²Dette er spesielt viktig når man bruker tjenester for å lagre repositories på nett, f.eks. [Github](#).

III .2 git init

`git` er basert på at man har en *oppbevaringsmappe* kalt repository. Om man ønsker å skrive kode i en mappe kalt `demo`, og at `git` skal følge med koden, kan man skrive følgende fra terminalen:

```
student@Ubuntu:~$ mkdir demo
student@Ubuntu:~$ cd demo
student@Ubuntu:~$ git init
```

```
Initialized empty Git repository in /home/student/demo/.git/
```

`mkdir` (*make directory*) vil opprette mappen `demo`, og `cd` (*change directory*) vil flytte brukeren inn i den. Til slutt vil kommandoen `git init` opprette en skjult mappe med navn `.git` inne i `demo`-mappen, som `git` vil bruke for å holde styr på filene i mappen. Det er viktig å nevne at `git init` må brukes for å opprette et *git directory* før man kan bruke de fleste kommandoene knyttet til `git`.

III .3 git status, git add, git commit

Dere har nå en tom `git`-mappe (`demo`). Kall kommandoen `git status`. Hvis dere ikke har gjort noen endringer i mappen så langt, vil dere få tilbake en melding som dette:

```
student@Ubuntu:~$ git status
```

```
On branch master
```

```
No commits yet
```

```
nothing to commit (create/copy files and use "git add" to track)
```

Stort sett er `git` veldig behjelpelig med å fortelle brukeren hva som foregår. Om man lurer på hva som skjer, er utskriften fra `git` oftest et godt svar.

Opprett nå en fil kalt `main.c` i `demo`-mappen, og skriv inn det følgende med en hvilken som helst tekstbehandler:

```
#include <stdio.h>

int main(){
    return 0;
}
```

Dersom dere nå lagrer denne filen, og kjører `git status` på nytt burde dere se noe slikt eller liknende avhengig av hvilken versjon av `git` dere har:

```
student@Ubuntu:~$ git status
```

On branch master

No commits yet

Untracked files:

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

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

Denne utskriften forteller oss at vi nå har en ny fil i demo-mappen ved navn `main.c`, som git foreløpig ikke bryr seg om. Kall nå `git add main.c`, etterfulgt av `git status` for å få opp denne meldingen:

```
student@Ubuntu:~$ git add main.c
student@Ubuntu:~$ git status
```

On branch master

No commits yet

Changes to be committed:

(use "git rm --cached <file>..." to unstage)
new file: main.c

Dette betyr at git har lagt til `main.c` i sitt *staging area*, som er stedet før git tar et bilde av mappen. Dersom man nå skriver `git commit -m "added main.c"` etterfulgt av `git lg`³ vil man se noe slikt:

```
student@Ubuntu:~$ git commit -m "added main.c"
```

```
[master (root-commit) 7cb84fb] added main.c
1 file changed, 5 insertions(+)
create mode 100644 main.c
```

```
student@Ubuntu:~$ git lg
```

```
* 7cb84fb (HEAD -> master) added main.c
```

Det siste vi ser på denne linjen er en stjerne. Denne stjernen representerer et bilde som git har tatt for oss. De sju heksadesimale tallene som følger etter er et utsnitt av en hash på 40 bokstaver, som git bruker for å identifisere bilder (eller commits). Denne hashen er spesielt viktig, om man vil inspisere tidligere bilder

³Dette er aliaset vi tidligere opprettet og defineres som `git log --all --oneline --graph --decorate`

som `git` har lagret.

I parentesen står det (`HEAD -> master`), som betyr at hodepekeren til `git` peker på akkurat dette *bildet*, som befinner seg på grenen `master`. Til slutt står det `added main.c` som er *commit*-meldingen vi ga dette bildet, for å beskrive hva vi har gjort.

III .4 Terminologi og gits indre

Figur 1 viser arbeidflyten som kan forventes med `git`,

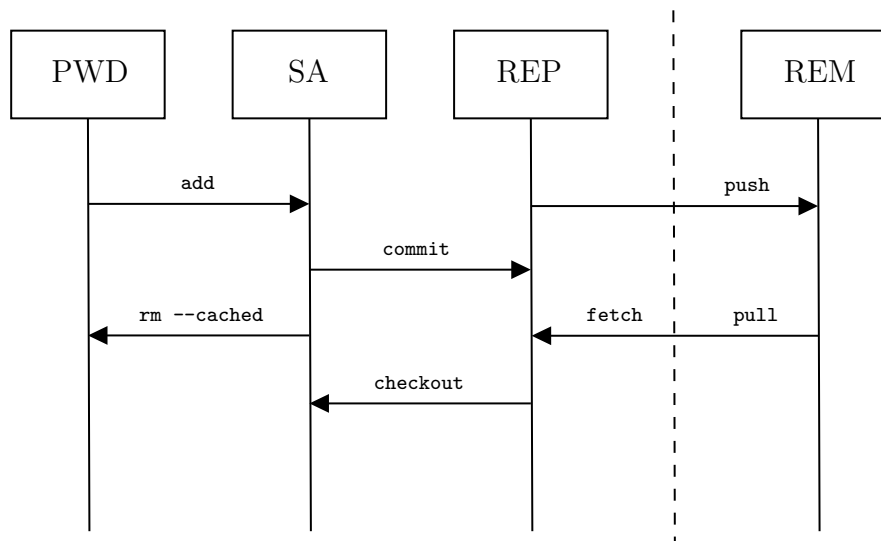


Figure 1: Arbeidsflyt i `git`.

hvor:

- **PWD**: *Current working directory*, dette er mappen som **git** holder styr på.
- **SA**: *Staging area*, her legges filer **git** skal *ta bilde av*, før de legges til i historikken.
- **REP**: *Repository*, dette er all historien **git** kjenner til. Bilder som er tatt av tidligere versjoner av filer legges til her, som en ny stjerne i en graf som representerer alt som har skjedd hittil.
- **REM**: *Remote*, dette er stort sett en ekstern server (men kan være en annen lokal mappe), dit **git** vil dytte lokale endringer, og ta endringer gjort av andre fra.

Til nå har vi vært innom **PWD**, **SA**, og **REP** ved at vi har laget en enkel C-kode (**main.c**) som vi har tatt *bilde av* med **git**. Vi kommer ikke til å sette opp eksterne servere, så vi kommer ikke borti **REM**⁴. For å få mer kunnskap om **git**, skal vi nå bygge videre på den lokale **git**-grafene vår.

III .5 git diff, git checkout

Endre **main.c** til å inneholde dette:

```
#include <stdio.h>

int main(){
    printf("Hello world\n");
    return 0;
}
```

Kjør deretter **git status**. Dere vil nå se:

```
student@Ubuntu:~$ git status
```

```
On branch master
```

```
Changes not staged for commit:
```

```
(use "git add <file>..." to update what will be committed)
```

```
(use "git restore <file>..." to discard changes in working directory)
```

```
modified:   main.c
```

```
no changes added to commit (use "git add" and/or "git commit -a")
```

Dette forteller oss at **git** vet at **main.c** har endret seg, men fordi vi ikke har lagt den til i *staging area*, vil ikke **git** ta et bilde av den.

⁴I praksis brukes ofte [Github](#) eller [Gitlab](#) som ekstern server for lagring av repositories.

For å få en oversikt over hva som har endret seg siden sist, kan man bruke kommandoen `git diff`. Dersom vi nå kaller `git diff main.c` vil vi få:

```
student@Ubuntu:~$ git diff main.c
```

```
@@ -1,5 + 1,6 @@
#include <stdio.h>\n
int main(){
+     printf("Hello world\n");
+     return 0;
}
```

hvor et pluss-tegn representerer en linje som har blitt lagt til, mens et minus-tegn representerer en linje som har blitt tatt bort. `git diff` er spesielt viktig dersom man har gjort mange endringer på en gang, og ikke husker hva man har gjort.

Dersom dere nå kjører `git add main.c` og `git commit -m "classic example code"`, etterfulgt av `git lg` vil dere se:

```
student@Ubuntu:~$ git add main.c
student@Ubuntu:~$ git commit -m "classic example code"
```

```
[master df7e5b2] classic example code
1 file changed, 1 insertion(+)
```

```
student@Ubuntu:~$ git lg
```

```
* df7e5b2 (HEAD -> master) classic example code
* 7cb84fb added main.c
```

Dette betyr at vi nettopp tok et nytt bilde av `main.c`, og at vi la denne til øverst i *historikktreet*. Den gamle koden som ikke gjorde noe ligger fortsatt i `gits` minne (med hashen `7cb84fb`), men grenen kalt `master` (og også vår hodepeker) peker til den nye `printf`-koden vi akkurat skrev som har fått hashen `df7e5b2`.

Det som gjør `git` veldig nyttig er at det er mulig å få tidligere *bilder*, ved å hoppe tilbake i historikk-grafen. Dersom dere nå kjører `git checkout 7cb84fb`⁵ vil dere få en melding som sier at dere er i `detached HEAD state`. Her kan man leke med den gamle koden som `git` har tatt vare på. Om man nå kaller `git checkout master` kommer man tilbake til den nye koden.

III .6 git branch, git merge

Når dere jobber på samme kodebase, kommer versjonskonflikter til å oppstå. Dette kan lett bli håndtert med `git branch` og `git merge`.

⁵Hashsummen deres kan være annerledes enn oppgaveteksten. Hashen til den gamle koden kan fås ved å kjøre `git lg`.

Kall først `git branch other`, etterfulgt av `git checkout other`. Dette vil lage en ny gren, kalt `other`, og hoppe til den. Denne grenen skal simulere at dere er to som jobber på samme kode. Dette er så vanlig at `git` har en innebygd kommando for akkurat dette: `git checkout -b <grennavn>`.

Om dere nå kaller `git lg`, vil dere se følgende:

```
student@Ubuntu:~$ git checkout -b other

Switched to branch other

student@Ubuntu:~$ git lg

* df7e5b2 (HEAD -> other, master) classic example code
* 7cb84fb added main.c
```

Nå har vi to grener som begge peker til den nyeste koden, men vi er på grenen `other`, og ikke `master`. La oss si at vi nå endrer på `main.c` slik:

```
#include <stdio.h>

int main(){
    printf("Hello world\n");
    printf("...and Mars\n");
    return 0;
}
```

Kjør så `git add main.c` og `git commit -m "greet mars as well"`. Dersom dere nå kjører `git lg` får dere:

```
student@Ubuntu:~$ git add main.c
student@Ubuntu:~$ git commit -m "greet mars as well"

[other eb331fb] greet mars as well
1 file changed, 1 insertion(+)

student@Ubuntu:~$ git lg

* eb331fb (HEAD -> other) greet mars as well
* df7e5b2 (master) classic example code
* 7cb84fb added main.c
```

Her ser vi at grenen `master` fortsatt ligger på koden med "Hello world", mens grenen `other` ligger på koden med "...and Mars".

Sett nå at dere er to som jobber i par, og at en har skrevet `world`-versjonen av koden og en har skrevet `Mars`-versjonen av koden. Dersom den som har skrevet `world`-versjonen nå skriver (bruk `git checkout master` for å bytte tilbake til

master-grenen):

```
#include <stdio.h>

int main(){
    printf("Hello world\n");
    if(1 > 0){
        return 1;
    }
    return 0;
}
```

Om man nå kjører `git add main.c`, `git commit -m "assert truth"`, og `git lg` får vi en historikkgraf som ser slik ut:

```
student@Ubuntu:~$ git add main.c
student@Ubuntu:~$ git commit -m "assert truth"

[master 835af8f] assert truth
1 file changed, 3 insertions(+)

student@Ubuntu:~$ git lg

* 835af8f (HEAD -> master) assert truth
| * eb331fb (other) greet mars as well
|/
* df7e5b2 classic example code
* 7cb84fb added main.c
```

Dette representerer at dere var enige på committen med hash `df7e5b2`, men at dere deretter har divergert til hver deres gren (nye `world`-grenen har fått hashen `835af8f`, mens `Mars`-grenen har fått hashen `eb331fb`). Dersom vi nå kaller `git merge other`, for å sammenslå `other`-grenen inn i `master`-grenen vil `git` klage med en feilmelding som burde seg noe slikt ut:

```
student@Ubuntu:~$ git merge other

Auto-merging main.
CONFLICT (content): Merge conflict in main.c
Automatic merge failed; fix conflicts and then commit the result
```

For å få mer informasjon, kan man kalle `git status`:

```
student@Ubuntu:~$ git status

On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
```

(use `"git merge --abort"` to abort the merge)

Unmerged paths:

(use `"git add <file>..."` to mark resolution)

both modified: main.c

no changes added to commit (use `"git add"` and/or `"git commit -a"`)

Her er git veldig hjelpelig og forteller brukeren nøyaktig hva som foregår: Vi holder på med en sammenslåing, men git vil ikke fullføre, fordi både `master` og `other` har endret på `main.c`.

For å fikse dette problemet kan man gjøre 2 ting:

1. Vi kan fikse konflikten ved at man endrer koden slik at den inneholder begge kodene og kjøre `git commit` for å manuelt fullføre sammenslåingen.
2. Vi kan kjøre `git merge --abort`, om vi ikke lenger vil slå sammen grenene.

Dersom man velger å gå for alternativ 1 siden vi helst vil beholde koden fra begge parter, kan man først åpne `main.c` på nytt for å se endringene som har blitt gjort (koden blir endret automatisk etter at man har kjørt `git merge`):

```
#include <stdio.h>

int main(){
    printf("Hello world\n");
<<<<<<< HEAD
    if(1 > 0){
        return 1;
    }
=====
    printf("...and Mars\n");
>>>>>>> other
    return 0;
}
```

Her kan man se at git automatisk har satt inn konfliktmarkører der koden var forskjellig. Alt mellom `<<<<<<< HEAD` og `=====` var på `master`-grenen, mens alt som ligger mellom `=====` og `>>>>>>> other` lå på `other`-grenen.

For å fortelle git at konfliktene er tatt hånd om, må man redigere filen slik den skal være, og så legge den til i git på vanlig måte. Dette gjøres ved å endre `main.c` til koden under:

```
#include <stdio.h>

int main(){
    printf("Hello world\n");
    printf("...and Mars\n");
}
```

```
    if(1 > 0){  
        return 1;  
    }  
    return 0;  
}
```

for å derette kjøre `git add main.c`, etterfulgt av `git commit -m "conflict solved"`. Hvis man nå kaller `git lg` har vi denne historikkgrafen:

```
student@Ubuntu:~$ git add main.c  
student@Ubuntu:~$ git commit -m "conflict solved"
```

```
[master 3bff360] conflict solved
```

```
student@Ubuntu:~$ git lg
```

```
* 3bff360 (HEAD -> master) conflict solved  
|\n| * eb331fb (other) greet mars as well  
* | 835af8f assert truth  
|/  
* df7e5b2 classic example code  
* 7cb84fb added main.c
```

Altså er kodebasene nå slått sammen, men som dere ser, vil ikke `other`- grenen automatisk trekkes etter. Denne grenen kan fjernes ved å kalle `git branch -d other`:

```
student@Ubuntu:~$ git branch -d other
```

```
Deleted branch other (was eb331fb).
```

```
student@Ubuntu:~$ git lg
```

```
* 3bff360 (HEAD -> master) conflict solved
|\
| * eb331fb greet mars as well
* | 835af8f assert truth
|/
* df7e5b2 classic example code
* 7cb84fb added main.c
```

III .7 git help

Dersom man er helt lost, så burde man bruke kommandoen `git help`. Dette er en kommando som er spesielt nyttig dersom man vil ha en oversikt over mulige flagg som enhver `git`-kommando støtter. I tillegg gir den litt informasjon om `git`-kommandoen selv. Eksempelsvis, `git help commit` får opp hjelpesiden til `git commit`.

1 Oppgave (50 %) - Grunnleggende Git

For å få godkjent øvingen, skal dere vise at dere har skjønnet det meste av walk-throughen. Dere skal derfor vise hva dere har gjort til en studass og besvare noen spørsmål fra studassen på sal, før dere får øvingen godkjent. Dere oppfordres også til å utforske `git` på egenhånd ved å bruke `git help` eller `git help tutorial`, ettersom dere kommer til å bruke `git` til heisprosjektet (og videre i arbeidslivet).

2 Oppgave (50 %) - GitHub

Når man bruker `git` i praksis, er det vanlig i kombinasjon med en såkalt kodevert-splattform. GitHub er det mest brukte eksempelet på dette, men det finnes også andre alternativer som Bitbucket, GitLab, m.m. I TTK4235 anbefaler vi bruk av GitHub, og som student ved NTNU får man faktisk tilgang til GitHub Pro. Her er det imidlertid verd å merke seg at bruk av GitHub kan medføre at alt du legger ut i repositoryet blir public, dvs. tilgjengelig for alle via f.eks. søk i Google. Dette er som regel ikke noe problem når du jobber med øvinger i et emne på NTNU (og definitivt ikke noe problem i TTK4235), men dersom du jobber i eller med en bedrift, eller prosjektet ditt av andre årsaker inkluderer konfidensiell kunnskap eller metodikk, kan det være lurt å tenke seg om to ganger. Dette gjelder også i tilfeller der du selv har en forretningside som du tenker du kanskje kan kommersialisere i fremtiden. I slike tilfeller kan det være lurt å finne en annen løsning enn

åpne repositories i GitHub.

For å ta i bruk GitHub må man først lage seg en bruker, og dette gjøres på nettsiden <https://github.com/>. Hvis man registrerer seg med stud-mailen gjør dette ting litt enklere når man skal oppgradere til Pro-bruker. Hvordan dette gjøres lar vi dere finne ut av på egenhånd.

Når man har laget seg en bruker og logget inn er det på tide å lage sitt første *repository*. Dette gjøres ved å trykke på den grønne knappen på hovedsiden, hvor det enten står **New** eller **Create repository**. Her får man muligheten til å velge navn under *Repository name* og beskrivelse under *Description*. Man får også muligheten til å bestemme om *repositoryet* skal være *Public* eller *Private*, altså om andre internett-brukere skal kunne se det eller ikke. Når man er fornøyd med valgene sine kan man trykke på **Create repository** for å fullføre prosessen. Deretter kommer man til hovedsiden til ditt nylagde *repository*.

Nå er det på tide å lage et såkalt *token*. Dette er et alternativ til å bruke passord til autentisering, når man bruker kommandolinja, altså terminalen, i Linux. Det finnes to typer *tokens*, nemlig *fine-grained personal access tokens* og *Personal access tokens (classic)*. Vi skal benytte oss av sistnevnte, ettersom de er bittelitt enklere å bruke. Et *token* knyttes typisk opp mot et *repository*, og brukes for å aksessere dette *repositoryet* fra terminalen. Følg disse stegene for å lage et *classic personal access token*:

1. Trykk på profilbildet ditt (øvre høyre hjørne), og deretter *Settings*.
2. Deretter trykker du på *Developer settings* i den venstre sidebaren.
3. Deretter trykker du på *Personal access tokens*, og så *Tokens (classic)* i den venstre sidebaren.
4. Trykk på *Generate new token*, og deretter *Generate new token (classic)*.
5. Under *Note*, skriv inn et navn til ditt *token*. Dette kan f.eks. være det samme som navnet på *repositoryet* du generelt kommer til å bruke det til.
6. Under *Expiration* velger du hvor lenge det skal være gyldig.
7. Under *Select scopes* velger du de rettighetene du vil at *tokenet* skal ha. Til våre formål i TTK4235 kan det være greit å ha så mange rettigheter som mulig, men dersom sikkerhet hadde vært av større viktighet, hadde vi begrenset dette også. Kryss derfor av i alle boksene.
8. Trykk på *Generate token* for å fullføre prosessen. Husk å kopiere *tokenet* et trygt sted, ettersom du ikke får muligheten til å se det igjen. Det er også alltid mulig å lage et nytt *token* dersom uhellet skulle være ute.

Nå er vi i stand til å klonere *repositoryet* vårt ved å bruke følgende kommando:
`git clone https://github.com/user_name/repository_name.git`, der "user_name" og "repository_name" byttes ut med henholdsvis GitHub-brukernavnet og *repository-navnet* ditt. Her vil du bli spurt om å skrive inn brukernavnet ditt og ditt

passord (her skriver du inn ditt *token*), og så har du klonet ditt *repository*.

Nå kan du for eksempel lage en enkel tekstfil i *repositoryet* via terminalen, og deretter bruke `git add file_name` for å legge til denne filen i *staging area*. Så kan du *committe* gjennom kommandoen `git commit -m "first commit"`, og deretter *pushe* ved kommandoen `git push`. Du har nå laget et GitHub-*repository*, og gjort en *commit*. Hvis du tar en titt på hovedsiden til ditt *repository* vil du nå se endringene som har blitt gjort. Arbeidsflyten ellers er lik som forklart tidligere i denne øvingen, bortsett fra at *merge* med fordel kan gjøres i nettsiden til selve *repositoryet*.

I dette eksempelet gjorde vi en *commit* rett inn i hovedgrenen `main`. Dette bør man egentlig aldri gjøre, og vi anbefaler å gjøre *commits* i andre grener, og deretter å *merge* i GitHub sin nettleser. Dette er en svært vanlig konvensjon som hovedsaklig går ut på å holde `main` kjørbare. I tillegg vil vi nevne at dere **ikke** bør bruke *global credentials* (f.eks. `--global`-flagget i `git config`) når dere jobber på offentlige datamaskiner, noe dere vil gjøre i TTK4235. For mer informasjon om Git og Git-konvensjoner anbefaler vi å ta en titt på boken *Pro Git* skrevet av Scott Chacon, som er tilgjengelig gratis på nett.

For å få denne oppgaven godkjent, vis hva dere har gjort til læringsassistentene.