Studenters arbeid med programmering i biovitenskapelige problemstillinger

En kvalitativ studie av biologistudenters arbeid med Python

Lars Erik Håland

Masteroppgave i biovitenskap
Institutt for biovitenskap
Det matematisk-naturvitenskapelige fakultet

UNIVERSITETET I OSLO

1. juni 2019

Studenters arbeid med programmering i biovitenskapelige problemstillinger

En kvalitativ studie av biologistudenters arbeid med Python i introduksjonskurs i programmering.

2019

Studenters arbeid med programmering i biovitenskapelige problemstillinger

Lars Erik Håland

http://www.duo.uio.no/

Trykk: Reprosentralen, Universitetet i Oslo

Abstract

Programming is getting a more prominent role in the natural sciences. This is visible through the renewal of curricula in the Norwegian school, as well as the demand from today's workplace. The University of Oslo (UiO) is among the first in Norway to include a mandatory programming course in their natural sciences. Since the research field of programming in natural sciences is still in its early stages, there is a demand for more research in this field. I have in my study tried to map how biology students work with programming tasks. The informants in this study are three groups of students who attend the course "Introduction to Computational Modelling in the Biosciences" (BIOS1100), which is a mandatory course for all students who study bioscience at UiO. The study has been conducted by using a combination of focus group interviews and observation of groups working with programming tasks. I have also used one question from a questionnaire that was given to all the students at BIOS1100.

The findings from this study show that students face several challenges while programming in bioscience. The challenges we found in this study include writing programs, understanding what the students are supposed to code, using different data structures and using mathematics. I argue that these challenges are caused by students lack in strategic knowledge in programming. In this study we mainly found two types of strategies used by the students; 1) use of example codes, 2) trying- and failing. We also see that tasks involving bioscientific knowledge did not affect how the students were writing their codes. We did however see that bioscience knowledge could have both a positive and negative effect on their ability to solve the programming problems. It seems though that having the bioscience included in the programming tasks makes the students more interested in learning how to program. In the end we can also conclude that the students are mostly programming in groups and this can have both positive and negative consequences for learning programming.

I have in this study concluded that problem-solving skills should have a bigger role in programming courses. "Computational thinking" may be a useful problem-solving strategy for the students to learn. In doing so they may get better and more efficient in solving their programming tasks.

Sammendrag

Programmering får en stadig sterkere rolle i de ulike realfagene. Dette ser vi gjennom fagfornyelsen i norsk grunn- og videregående skole, og på behovet i næringslivet. Universitetet i Oslo er blant de første i Norge til å innføre programmering som en obligatorisk del av realfagene. Siden programmering i realfagsutdanning er så nytt er det behov for forskning på dette feltet. I min studie har jeg forsøkt å kartlegge hvordan biologistudenter arbeider med programmeringsoppgaver. Informantene i denne studien har vært tre grupper med studenter som tar emnet «Innføring i beregningsorienterte modeller i biovitenskap» (BIOS1100), et obligatorisk emne for alle studenter som studerer biovitenskap ved UiO. Studien har blitt gjennomført ved å bruke en kombinasjon av fokusgruppeintervju og observasjon av gruppearbeid. I tillegg har jeg anvendt et spørsmål i et spørreskjema som ble gitt ut til alle studentene ved BIOS1100.

Funnene fra denne studien viser at studentene møter en rekke utfordringer når de programmerer i biovitenskapelige emner. Utfordringene er i stor grad knyttet til å lage store programmer, forstå hva som skal programmeres, bruke ulike programstrukturer og anvende matematikk. Jeg mener at mange av disse utfordringene er en konsekvens av manglende strategisk kunnskap hos studentene. I hovedsak viser studentene til to typer strategier når de programmerer; 1) bruk av eksempelprogrammer, 2) prøving og feiling. Videre ser vi at biovitenskapelige problemstillinger har liten påvirkning på arbeidet med å skrive programmer, men at det både kan ha positiv og negativ effekt på studentenes ferdigheter til å løse programmeringsoppgaver. Samtidig ser studentene ut til å ha større interesse for programmering når de får arbeide med biovitenskapelige problemstillinger. Til slutt ser vi at studentene for det meste programmerer i grupper og at dette kan ha både positive og negative følger for studentenes læring av programmering.

I denne studien argumenterer jeg for at problemløsning bør vektlegges mer i programmeringskurs. Her kan «Computational thinking» være en nyttig problemløsningsstrategi for å lære studentene hvordan de bedre og mer effektivt løser programmeringsoppgaver.

Forord

Det er utrolig å tenke på at min fem år lange utdanning er over. Det er med både glede og sorg at jeg nå sier farvel med Oslo, Blindern og de vennene jeg har fått her. Da jeg startet med denne masteroppgaven for halvannet år siden, visste jeg ikke hva jeg gikk til. Det har vært både spennende og lærerikt å få arbeide med et forskningstema som er så nytt. Jeg har forsket på programmering, en fagdisiplin jeg ikke har vært borti siden først semester i min Lektorutdanning. Allikevel har jeg kommet i mål og året har gått mye bedre enn forventet. Spesielt har jeg har lært mye om hvordan studenter arbeider med programmering, en erfaring jeg setter stor pris på når jeg nå skal ut og undervise om dette selv.

Først vil jeg rette en takk til mine tre flinke veiledere. Ellen Karoline Henriksen har vært en uvurderlig hjelp med både metodevalg og analyse, og har gitt meget gode tilbakemeldinger gjennom hele oppgaven. Lex Nedebragt har vært til stor hjelp med å velge programmeringsoppgavene og har vært en flott ressurs når jeg har skrevet om programmering. Spesielt vil jeg rette en takk til min hovedveileder Tone Fredsvik Gregers. Ikke bare har hun vært til stor hjelp med masteroppgaven, men hun har også bidratt med private kollokvier for å forberede meg og min medstudent til en eksamen i spesialpensum vi har måtte hatt. Hvordan hun får tid til alt, er forbi min forståelse.

Jeg vil også takke min medstudent, Marthe Mjøen Berg. Hadde det ikke vært for henne ville jeg aldri valgt denne masteren. Dette året hadde ikke vært det samme om det ikke var for kaffepauser, fremtidsdrømmene, frustrasjon og tørre vitser. Til slutt fortjener min samboer Eva Mørch en takk for den forståelsen hun har vist mens jeg har skrevet denne oppgaven og for hjelpen hun har gitt med korrekturlesing.

Innholdsfortegnelse

1 Innledning

1.1 Bakgrunn og motivasjon

Programmering er et fagfelt som har hatt enorm påvirkning på hverdagen til mennesker. Uten programmering ville vi hverken hatt internett eller smarttelefoner. Samtidig har det vært en viktig faktor for den utviklingen vi har hatt innen en rekke fagområder som både matematikk, fysikk og biologi. Fra et undervisningsperspektiv har programmering tradisjonelt blitt behandlet som et eget fag. Nå skjer det derimot en endring i det norske samfunn, hvor det i 2020 blir det en fagfornyelse i norsk grunnskole og videregående opplæring (Kunnskapsdepartementet, 2018). Fag som naturfag og matematikk får nye læreplaner og til tross for at innholdet i læreplanene ikke er fastsatt, har Kunnskapsdepartementet etablert noen kjerneelementer som skal være med. Kjerneelementene er de mest sentrale temaene som elevene skal kunne etter endt utdanning i grunnskole eller videregående opplæring, og innholdet i læreplanen blir bygget opp rundt dem. Nytt for matematikk og naturfag er blant annet at programmering nå skal integreres i fagene. Ved å gjøre programmering obligatorisk for alle elever i norsk skole, sender fagfornyelsen et signal om at programmering er en ferdighet alle norske elever bør ha.

Programmering har stor påvirkning på biologifaget, men utdanningen av biologer har ikke klart å følge opp denne utviklingen (Pevzner & Shamir, 2009). Universitetet i Oslo (UiO) er derimot en av få institusjoner som i løpet av de siste 10 årene har integrert programmering som en obligatorisk del i de naturvitenskapelige emnene. Dette har de gjort ved å innføre programmering i første semester for alle som studerer realfag. For biologi opprettet universitetet høsten 2017 emnet «Innføring i beregningsmodeller for biovitenskap» (BIOS1100) for alle som studerer biovitenskap eller går lektorstudiet i biologi/kjemi. Bakgrunnen for dette er at universitetet ønsker å gi studentene en utdanning som samsvarer med dagens arbeidsliv, men også at programmering kan bidra til bedre forståelse av naturvitenskapelige problemstillinger (Malthe-Sørenssen, Hjorth-Jensen, Langtangen, & Mørken, 2015).

Med økt fokus på realfaglig programmering i alle utdanningsnivåene er det viktig å forstå hvordan studenter arbeider når de programmerer. Siden realfaglig programmering er relativt nytt er det foreløpig lite forskning på programmering i realfaglig utdanning i norsk

sammenheng. Gjennom denne masteroppgaven ønsker jeg å sette fokus på programmering i realfaglige emner og gjør dette ved å se på programmering i biologi.

1.1.1 Emnet BIOS1100

BIOS1100 har som mål å gi en innføring i å lage og eksperimentere med enkle modeller av biologiske systemer. Læringsmålene til BIOS1100 er hentet fra emnesiden på nett (Universitetet i Oslo, 2018):

Etter å ha fullført emnet:

- har du fortrolighet med Python-programmering og kan bruke datastrukturer, funksjoner og moduler, samt løkker og betingelsestester
- kan du organisere biologiske data, lese og skrive slike data til/fra fil og lage grafiske framstillinger
- kan du modellere biologiske systemer med hjelp av vektor- og matrisearitmetikk og grunnleggende sannsynlighetsregning
- kan du dokumentere, presentere og formidle enkle modeller av biologiske systemer

Programmeringsspråket som blir brukt i BIOS1100 er Python. Python er et imperativt, høynivå, fortolket og objektorientert programmeringsspråk (Dvergsdal, 2017). Det brukes ofte i nybegynnerkurs i programmering, siden det er forholdsvis lett å lære de elementære programmeringskonseptene i Python. Undervisningen på BIOS1100 var lagt opp med to timer forelesning i uka og to timer ekstra undervisning for studenter som ønsket mer hjelp i emnet. I tillegg hadde studentene fire timer sammenhengende gruppeundervisning per uke. Gruppetimene var obligatoriske, og studentene deltok i samme gruppetime hver gang. Gruppetimene var som regel delt i to deler. I første del av gruppetimen ble studentene introdusert for spesifikke temaer i programmering gjennom forskjellige studentaktive undervisningsopplegg. Hovedsakelig ble det undervist gjennom livekoding, som bestod av at en gruppelærer skrev programmene på en prosjektor foran studentene. Studentene satt med egne datamaskiner og programmerte sammen med gruppelæreren. Den andre delen av gruppetimen var satt av til at studentene kunne arbeide med programmeringsoppgaver samtidig som de hadde tilgang på hjelp fra gruppelærer. Dette var i alle fall hensikten med

emnet, men ofte ble livekodingen såpass tidkrevende at mesteparten av gruppetimen gikk til det. På hver gruppetime var det mellom 40 og 50 studenter og de arbeidet i faste grupper på 4-8 studenter gjennom hele semesteret. Hver gruppe hadde tilgang på en felles dataskjerm der en av studentene kunne koble opp sin egen datamaskin når de arbeidet med programmeringsoppgaver.

Sluttvurderingen på emnet var en digital eksamen i slutten av semesteret. Studentene hadde mulighet til å skrive programmene, men ikke til å kjøre programmene for å se om de fungerte.

1.1.2 Forskningsspørsmål

Hensikten med denne oppgaven er å få innblikk i hvordan studentene i BIOS1100 arbeider med programmeringsoppgaver. Siden programmering i biologiutdanning er ett såpass nytt felt, ønsker jeg å se hvilken påvirkning det har på studentenes arbeid når de må ta hensyn til biovitenskapelige problemstillinger mens de programmerer.

Min problemstilling i denne oppgaven er:

Hvordan arbeider studenter når de programmerer i introduksjonskurs i biologi?

For å finne svar på denne problemstillingen har jeg stilt følgende forskningsspørsmål:

1. Hvilke utfordringer møter studentene når de programmerer?
2. Hvilke strategier bruker studentene når de programmerer?
3. Hvilken rolle spiller det biologifaglige for studentene når de programmerer?
4. Hvilke arbeidsvaner har studentene med programmeringsoppgaver i emnet BIOS1100?

Datamaterialet jeg bruker i denne oppgaven er observasjon av tre grupper som løser et oppgavesett med programmeringsoppgaver i BIOS1100. De tre gruppene har også blitt intervjuet i etterkant av deres arbeid med oppgavesettet. Som et supplement til dette datamaterialet, har jeg også analysert ett spørsmål fra et spørreskjema som ble laget i samarbeid med en annen masterstudent, Marthe Mjøen Berg og som ble delt ut til alle studentene i BIOS1100 i siste forelesning.

1.1.3 Avgrensning av oppgaven

Gitt studiens omfang og metodikk, er det noen aspekter jeg ikke kommer til å undersøke i denne oppgaven. Jeg kommer ikke til å se på studentenes læringsutbytte. For å gjøre dette måtte jeg ha sammenliknet resultatene mellom studenter på en slutteksamen, samtidig som jeg innførte en målbar endring i undervisningen til studentene.

Jeg kommer heller ikke til å gå i dybden av det programmeringstekniske. Dette er fordi denne studien ser på hvordan studentene arbeider når de programmerer og ikke på selve programmeringen. Designet av oppgaven forhindrer også dette, da jeg måtte ha gjort et videopptak eller tatt bilde av studentens programmer for å studere hvordan de skriver programmene.

BIOS1100 er et beregningsorientert emnet i biologi. Til tross for dette er det selve programmeringsdelen jeg først og fremst ser på i denne masteroppgaven. Derfor kommer jeg i hovedsak til å omtale programmering i resten av oppgaven, og ikke beregningsorientert biologi.

2 Teori og tidligere forskning

I dette kapittelet vil jeg starte med å legge frem det teoretiske rammeverket som jeg bruker i min studie. Her presenterer jeg en modell av Computational Thinking (CT) som det teoretiske rammeverket i min oppgave og noen sammenhenger mellom CT og problemløsning. Deretter vil jeg presentere forskning som omhandler CT og introduksjonskurs i programmering. På slutten av kapittelet vil jeg presentere forskning rundt studenters forkunnskaper i programmering, programmering i en faglig kontekst og bruk av eksempler i undervisningen.

2.1 Computational thinking – Modell og definisjon

Begrepet «Computational thinking» bygger på arbeidet til Semour Papert (Papert, 1980), men ble popularisert av Jeanette M. Wing i 2006. CT handler om å løse problemer og designe systemer ved å bygge på konsepter fra informatikken (Wing, 2006). CT beskriver de tankeprosessene som inngår i å formulere et problem til en beregningsorientert løsning, som så kan løses med eller uten en datamaskin (Wing, 2011)

De siste årene har CT vært et mye omdiskutert tema i programmeringsutdanningen. En rekke operasjonaliseringer av CT har blitt brukt i ulike studier (Barr, Harrison, & Conery, 2011; Berland & Wilensky, 2015; Weintrop *et al.*, 2015), men det er ikke blitt etablert en felles definisjon på hva CT egentlig er (Barr *et al.*, 2011; Grover & Pea, 2013). I min masteroppgave har jeg valgt å bruke en definisjon laget av Shute, Sun, og Asbell-Clarke (2017):

«the conceptual foundation required to solve problems effectively and efficiently (i.e., algorithmically, with or without the assistance of computers) with solutions that are reusable in different contexts» (Shute *et al.,* 2017, s.5).

CT defineres som et fundament for å løse problemer på en effektiv måte, med eller uten datamaskiner. Løsningene som lages gjennom CT skal kunne videreføres og brukes for å løse problemer i lignende situasjoner. For å beskrive hva som inngår i å løse problemer gjennom CT, har Shute *et al.* laget en modell som beskriver seks komponenter vi kan dele CT inn i. Som tabell 1 viser, kan CT deles opp i: *problemnedbrytning, abstraksjon, algoritmer, feilsøking, iterasjon og generalisering*. Til hver komponent er det beskrevet hvilke kognitive prosesser som ligger til grunn for dem. Modellen er oversatt fra engelsk til norsk.

Tabell 1 – Beskrivelse av komponentene i CT og deres kognitive prosesser fra Shute et al. (2017)

CT komponenter	**Underliggende kognitive prosesser**
Problemnedbrytning	Dele opp et komplekst problem/system til håndterlige deler. De ulike delene er ikke tilfeldige, men funksjonelle elementer som kollektivt utgjør hele problemet/systemet.
Abstraksjon	Ekstrahere essensen av et (komplekst) system. Abstraksjon har tre underkategorier: 1. *Datainnsamling og analyse:* Samle inn den mest relevante og viktige informasjonen fra flere kilder og forså forholdet mellom de ulike datasettene. 2. *Gjenkjenne mønstre:* Identifisere mønstre/regler som ligger i dataene/informasjonen. 3. *Modellering:* Bygge modeller eller simuleringer som representerer hvordan et system opererer, og/eller hvordan et system vil fungere i fremtiden.
Algoritmer	Designe logiske og strukturerte instruksjoner for å gi en løsning til et problem. Instruksjonene kan utføres av en datamaskin eller et menneske. Det er fire underkategorier: 1. *Algoritmisk design:* Lage et sett med strukturerte steg for å løse et problem. 2. *Parallellisme:* Utføre flere steg på samme tidspunkt. 3. *Effektivitet:* Designe færrest mulige steg for å løse et problem ved å fjerne overflødige og unødvendige steg. 4. *Automatikk:* Automatisere utførelsen av prosedyren når det kreves for å løse liknende problemer.
Feilsøking	Detektere og identifisere feil og fikse feilen når løsningen ikke virker som den skal.
Iterasjon	Gjenta designprosessen for å raffinere løsningen til et ideelt resultat er oppnådd.
Generalisering	Overføre CT ferdigheter til et bredt spekter av situasjoner/domener for å løse problemer effektivt.

Modellen til Shute *et al.* legger vekt på at det er en systematisk måte å løse problemer. Kort forklart innebærer det at man bruker *problemnedbrytningen* for å bryte ned et komplekst problem i mindre deler og deretter løse hver del på en systematisk måte. *Abstraksjon* handler om å finne trender i dataene og setter parametere for løsningen. Designet av *algoritmer* lager et verktøy som kan løse problemet, mens *itererende*, systematisk *feilsøking* bidrar til at hver enkelt del løses effektivt. Fra abstraksjonen av data kan man *generalisere* løsninger som kan brukes til lignende problemer.

Det er to grunner til at jeg velger å bygge mitt teoretiske rammeverk på Shute *et al.* sitt arbeid. For det første er definisjonen til Shute *et al.* et resultat av en litterær metaanalyse av forskningsartikler som er relevant for CT. Deres definisjon er et resultat av de vanligste karakteristikkene og komponentene som benyttes om CT. Ved å bruke definisjonen til Shute *et al.* (2017) i min masteroppgave vil jeg bygge på en beskrivelse av CT som er godt forankret i forskningslitteraturen. I tillegg ønsker jeg med dette arbeidet å bidra til en felles konsensus for hva CT er både innenfor didaktikken og forskningen. Den andre grunnen er at Shute *et al.* har laget denne definisjonen med den hensikt å bistå den pedagogiske utviklingen av CT i skolen. Definisjonen er først og fremst laget for elever i aldersgruppen 5-18 år, men kan også brukes i undervisningen på bachelornivå (Shute *et al.*, 2017).

2.2 Eksisterende forskning om læring av programmering

Siden definisjonen av CT til Shute *et al.* (2017) er såpass ny er det foreløpig lite teori som omhandler dette temaet. Min masteroppgave vil derfor ha sin teoretiske tyngde fra tidligere forskning på programmering. Her vil jeg se på forskning knyttet til komponentene CT består av. De to siste komponentene av CT, iterasjon og generalisering, har jeg derimot ikke sett på i denne studien så de blir ikke omtalt videre i denne oppgaven.

2.2.1 Computational thinking og problemløsning

Begrepet «problemløsning» har ulike definisjoner i litteraturen. George Pólya definerte problemløsning allerede i 1945, da han beskrev det gjennom 4 trinn (Pólya, 1945):

1. Understand
2. Plan
3. Carry out
4. Look back

I dag anser vi problemløsning litt mer avansert enn denne versjonen, men prinsippene bygger på arbeidet til Pólya. I modellen til Shute *et al.* (2017) ser vi flere paralleller til Pólyas definisjon. Abstraksjon og problemnedbrytning handler om å forstå hva oppgaven spør om, dele det opp i håndterlige deler og å planlegge utførelsen av oppgaven. Lage og programmering av algoritmer er å utføre planen, mens ved feilsøking, iterasjon og

generalisering ser man tilbake på løsningen, finpusser og overføre dette til andre lignende problemer.

Når det kommer til programmering ansees problemløsning som en nødvendig ferdighet studentene bør besitte (de Lira Tavares, de Menezes, & de Nevado, 2012). Til tross for at problemløsning bør stå sentralt i programmeringskurs, viser forskningslitteraturen at mange studenter mangler tilstrekkelige problemløsningsstrategier (Medeiros, Ramalho, & Falcao, 2018). Studenter som lærer seg programmering opplever ofte utfordringer når de skal analysere et problem, planlegge og lage løsninger. (Hazzan, Lapidot, & Ragonis, 2011). Hazzan *et al.* mener dette kommer av at introduksjonskursene fokuserer mest på å lære syntaktisk kunnskap og lite på å lære problemløsning. Når studentene ikke lærer problemløsning, utvikler de ofte egne, ineffektive strategier. Disse strategiene er ofte dominert av en prøv- og feil-mentalitet der studentene prøver seg frem når de lager programmer (Hazzan *et al.*, 2011). En norsk studie av fysikkstudenter som programmerte på UiO viste at nettopp dette var en av utfordringene studentene møtte. Når studentene ble overlatt til å arbeide med programmeringsoppgaver på egenhånd, ble det ofte brukt mye prøv- og feil-taktikker i oppgavene (Sørby & Angell, 2012). Sørby og Angell mener at studentene må få tilstrekkelig oppfølging slik at de får et mer bevisst forhold til hvilke strategier de bruker når de programmerer. De funnene som er presentert i dette avsnittet indikerer at programmeringskurs bør ha fokus på å undervise problemløsningsstrategier, som for eksempel CT.

Noen av utfordringene med CT er at det mangler en felles enighet om en definisjon, men også at det fortsatt er i et tidlig stadium innenfor forskningsfeltet (Shute *et al.*, 2017). Det er også vanskelig å sammenlikne CT mellom forskjellige fagfelt siden dette har fått så lite fokus i forskningen (Czerkawski & Lyman, 2015). Dette må derfor tas med i betraktningen når jeg drøfter resultatene i min studie. Samtidig viser det at det er et behov for å forske mer på CT både i biologi og programmering.

Det er ikke gjort mange studier på sammenhengen mellom læringsutbytte og anvendelse av CT i undervisningssituasjoner. Samtidig henviser Weintrop *et al.* (2015) til noen studier som indikerer at gjennomtenkt bruk av CT i undervisningen kan føre til dypere læring av realfaglige temaer (Repenning, Webb, & Ioannidou, 2010; Sengupta, Kinnebrew, Basu, Biswas, & Clark, 2013; Wilensky, Brady, & Horn, 2014). En studie har undersøkt effekten av å introdusere CT tidlig i nybegynnerkurs i programmering (Fernández, Zúñiga, Rosas, &

Guerrero, 2018). Undersøkelsen viste at CT hadde positiv effekt på studentenes ferdigheter innen abstraksjon, problemnedbrytning, anvendelse av algoritmer, og deres generelle problemløsningsferdigheter.

CT kan undervises på forskjellige måter. Hsu, Chang, og Hung (2018) har gjort en gjennomgang av forskningslitteraturen og sett hvilke typer læringsstrategier som ofte brukes for å promotere CT. Det ble identifisert 16 forskjellige måter å undervise CT på, men siden min studie undersøker studenter på universitetsnivå, forholder jeg meg til de strategiene som er mest omtalt på det nivået. Hsu *et al.* (2018) fant at problembasert læring, prosjektbasert læring og samarbeidende læring var mest omtalt for å fremme CT på universitetsnivå. Problembasert læring handler om å bruke problemløsning for å fremme kunnskap og forståelse (Wood, 2003). Det er ikke problemløsning i seg selv som er viktig, da det handler om å bruke riktige typer problemer for å trigge studentenes læring. Ifølge Wood er også gruppearbeid en viktig del av problembasert læring ettersom det bidrar til å utvikle kunnskap, ansvar for egen læring, problemløsning og samarbeid. Prosjektbasert læring handler på sin side om at studentene lærer gjennom prosjekter (Jones, Rasmussen, & Moffitt, 1997). Denne formen for arbeid involverer ofte komplekse spørsmål eller problemer som krever at studentene må designe og løse problemer, gjøre ulike valg, eller undersøke forskjellige aktiviteter. Samarbeidende læring krever at studentene arbeider sammen om å løse problemer (Dillenbourg, 1999). Dette kan gjennomføres på to måter. Studentene kan løse deler av et problem hver for seg og så sette sammen delene til en løsning, eller så kan de løse hele problemet sammen.

2.2.2 Problemnedbrytning og abstraksjon – Starten på problemløsning

Problemnedbrytning inngår som en viktig del av problemløsning og ansees som en nøkkelferdighet for programmering (Pearce, Nakazawa, & Heggen, 2015). Det er også den første komponenten i modellen til Shute *et al.* (2017). Når vi bryter ned et problem deler vi det opp i mindre delproblemer og da er det noen ting man må ta hensyn til (Liskov & Guttag, 2001). Delproblemene må være delt opp sånn at de fungerer på samme detaljnivå i oppgaven og hvert delproblem må kunne løses individuelt. Hver av løsningene på delproblemene må kunne settes sammen til en komplett løsning på det opprinnelige problemet. Samtidig ser man at studenter som lærer seg å programmere ofte sliter med å identifisere de ulike

komponentene av problemet og de relevante algoritmiske strukturene som må inn i programmet (Keen & Mammen, 2015; Raadt, Toleman, & Watson, 2004).

Det er ofte en antakelse at studentene i introduksjonskurs behersker problemnedbrytning og derfor blir ikke dette undervist i kurset. Konsekvensen blir at studentene sliter med å forstå hvordan de skal starte å løse større problemer når de programmerer (Keen & Mammen, 2015). To studier har vist at studentene i introduksjonskurs generelt blir bedre til å bryte ned problemer hvis dette blir integrert i undervisningen (Muller, Ginat, & Haberman, 2007; Pearce *et al.*, 2015). De to studiene har brukt ulike undervisningsopplegg for å introdusere problemnedbrytning og at det er brukt forskjellige tilnærminger indikerer at det ikke kun finnes én korrekt måte å lære studentene problemnedbrytning. Det virker som det er viktigere å bevisstgjøre studentene på hvordan de kan anvende problemnedbrytning og la dem få arbeide aktivt med dette.

Wing (2011) mener at abstraksjonsprosessen er den viktigste delen av CT. Abstraksjon gjør det mulig å håndtere komplekse problemer ved at man tar ut essensen av problemet og ignorerer irrelevant informasjon. Dette gjøres gjennom analyse og identifisering av trender i dataene (Shute *et al.*, 2017). Når man arbeider med abstraksjon, arbeider man normalt med flere nivåer av dette (Wing, 2011). I Python ligger det for eksempel inne egne kommandoer som utøver spesifikke funksjoner. Et eksempel på dette er *Choice* funksjonen, som når brukt riktig i et program lager et tilfeldig produkt fra de verdiene som settes inn i funksjonen. Når studentene arbeider med *Choice* arbeider de i et høyere abstraksjonsnivå, siden de ikke trenger å ta hensyn til hvordan *Choice* virker, bare hva den gjør.

En studie gjort av Haberman og Muller (2008) har avdekket flere typer utfordringer studenter møter når de arbeider med abstraksjon. I deres studie observerte de en korrelasjon mellom studentenes manglende evne til å ignorere irrelevant informasjon i programmering og deres utfordringer med å finne riktig svar på et problem. De observerte også at studentene slet med å bevege seg mellom de ulike abstraksjonsnivåene. Når det kommer til å løse problemer sliter ofte nybegynnere i programmering å forstå hva oppgaven ber dem om å gjøre (Muller, 2005; Robins, Rountree, & Rountree, 2010). I en annen studie observerte Muller (2005) at studentene ofte ble distrahert av den overfladiske informasjonen i problembeskrivelsen og at dette gjorde det vanskeligere for dem å løse problemet.

Abstraksjon er derimot ikke alltid et enkelt konsept for studentene å forstå (Hazzan & Kramer, 2007; Qualls, Grant, & Sherrell, 2011). I følge Hazzan og Kramer skyldes dette at abstraksjon nettopp er et så abstrakt begrep. Abstraksjon kan ikke karakteriseres ved bestemte regler eller spesifikke definisjoner, ei heller kan det presenteres gjennom spesifikke temaer i programmeringen. Qualls *et al.* (2011) observerte også i sin studie at studenter i programmeringskurs ofte har tilstrekkelige kunnskaper til å anvende abstraksjon, men at de ikke har en klar forståelse av hva abstraksjon faktisk er.

2.2.3 Algoritmer og ulike kunnskapsformer i programmering

De kognitive prosessene bak Algoritmer i CT innebærer å designe et sett med logiske og strukturerte steg for å løse et problem (Shute *et al.*, 2017). Dette innebærer at man både må kunne designe en algoritme og at man lager et program som kjører algoritmen. Det er gjort en del forskning på hvilke utfordringer nybegynnere møter når de skal programmere (Medeiros *et al.*, 2018; Qian & Lehman, 2017). I forskningslitteraturen har det blitt brukt ulike rammeverk for å kategorisere de ulike utfordringene studentene møter. En måte å kategorisere utfordringer på er å knytte dem til den kunnskapene som kreves for å programmere (Qian & Lehman, 2017). Læring av programmering kan knyttes til at det oppstår kognitive endringer i den som lærer (Bayman & Mayer, 1988). Gjennom de kognitive endringene vil man tilegne seg kunnskaper som vi deler inn i de tre kategoriene: *syntaktisk kunnskap, konseptuell kunnskap* og *strategisk kunnskap* (Bayman & Mayer, 1988).

Syntaktisk kunnskap er å kjenne til egenskapene og reglene til et bestemt programmeringsspråk (Bayman & Mayer, 1988). Et eksempel på en syntaksregel i Python er at man må ha et kolon bak en *if*-kommando. De vanligste feilene nybegynnere gjør når de programmerer er knyttet til syntaksfeil. En studie gjort av Altadmri og Brown (2015) viste at de vanligste syntaksfeilene blant nybegynnere var feil plassering av parenteser eller anførselstegn i programmet. Samtidig som syntaksfeil er de vanligste feilene blant nybegynnere, er det denne typen feil de bruker minst tid på å rette opp (Altadmri & Brown, 2015). I sin litteraturgjennomgang viser Medeiros *et al.* (2018) at syntaks er den utfordringen som oftest blir nevnt i forskningsartikler om introduksjonskurs i programmering. At syntaksfeil nevnes flest ganger trenger ikke å bety at studentene opplever det som mest utfordrende, men at det er en utfordring som er vanlig blant studentene.

Konseptuell kunnskap handler om hvordan ulike strukturene og prinsippene virker i et program (Bayman & Mayer, 1988). I Python er dette for eksempel å forstå hvordan en liste lagrer informasjon og hvordan listen må plasseres i programmet for å fungere. Utfordringer knyttet til konseptuell kunnskap kan ofte skyldes feil i programmererens mentale modeller (Bayman & Mayer, 1988). Programmererens mentale modeller er deres forståelse av de «usynlige» prosessene som skjer når vi kjører programmet (Bayman & Mayer, 1983). Det handler om hvordan programmereren ser for seg at hver enkel del av programmet virker. Når programmereren har misoppfattelser i sine mentale modeller, kan dette føre til vanskeligere og mer forankrede feil enn for syntaktiske feil (Bayman & Mayer, 1983). Mye av forskningslitteraturen viser også at studentene sliter med å forstå de ulike programstrukturene i et program og hvordan programmet skal settes sammen for å fungere (Medeiros *et al.*, 2018).

Strategiske kunnskaper i programmering er kunnskap som kreves for å planlegge, skrive og feilsøke i programmer (McGill & Volet, 1997). Dette innebærer å ha de nødvendige syntaktiske og konseptuelle kunnskapene for å programmere, i tillegg til å ha de nødvendige strategiene for å skrive programmet. Nybegynnere som ikke har tilstrekkelige syntaktiske og konseptuelle kunnskaper vil derfor møte utfordringer når de skal løse et programmeringsproblem (Qian & Lehman, 2017). Flere studier viser også at nybegynnere ofte har dårlige eller utilstrekkelige strategier for å løse programmeringsoppgaver (Clancy & Linn, 1999; Lister, Simon, Thompson, Whalley, & Prasad, 2006).

2.2.4 Feilsøking

Feilsøking er definert i modellen til Shute *et al.* (2017) som en kognitiv prosess der man identifiserer og reparerer feil når løsningen ikke virker som den skal. Som nevnt i avsnitt 2.2.3 kan feil være enten syntaktiske eller konseptuelle. Hvis studentene gjør en syntaksfeil i Python får de opp en feilmelding når de forsøker å kjøre programmet. Feilmeldingen gir en indikasjon på hva som er feil og hvor i programmer feilen ligger. En studie av Kummerfeld og Kay (2003) har sett på hva som kjennetegner nybegynnere og erfarne programmerere når de feilsøker og reparerer syntaksfeil. For å korrigere syntaksfeil mest effektivt, er det viktig at programmereren benytter ulike strategier (Kummerfeld & Kay, 2003). Erfarne programmerere bruker ofte spesifikke strategier når de løser syntaksfeil som er kjent for dem. Dette innebærer for eksempel at når det kommer opp en feilmelding, vurderer de feilmeldingen og ser på

programmet som en helhet før de retter opp feilen. Nybegynnere som møter kjente feilmeldinger vil kunne ha en tendens til å gå rett inn i programmet, korrigere feilen, og kjøre programmet på nytt. Denne tilnærmingen vil ofte reparere feilen, men om programmet i sin helhet gir feil svar kan det bli mer krevende å rette opp programmet. Dukker det opp ukjente feilmeldinger bruker erfarne programmerere ofte mer generelle, heuristiske strategier (Kummerfeld & Kay, 2003). Når nybegynnere møter ukjente feilmeldinger vil de ofte spørre om hjelp, eller ty til ulike former for hjelpemidler. Felles for både nybegynnere og erfarne programmerere er at de ofte vil prøve seg frem om de møter ukjente feilmeldinger de ikke klarer å fikse (Kummerfeld & Kay, 2003).

Konseptuelle feil kan lede til større utfordringer enn hva syntaksfeil gjør. Det kan være vanskelig for studentene å forstå de konseptuelle feilene og de er ofte vanskeligere å korrigere enn syntaksfeil (Bayman & Mayer, 1983). Som for syntaksfeil, kan også konseptuelle feil gi feilmeldinger når studentene programmerer i Python. En utfordring med konseptuelle feil kan derimot være at programmet virker, men at det produserer et annet resultat enn forventet. Denne formen for feil kalles logiske feil (Hristova, Misra, Rutter, & Mercuri, 2003). I en studie gjort av Ettles, Luxton-Reilly og Denny (2018) ble logiske feil delt inn i tre kategorier: *algoritmiske, feiltolkninger* og *misoppfatninger*. Algoritmiske feil kommer av at algoritmen som brukes er feil. Feiltolkninger er når studentene feiltolker spørsmålet og gjør programmeringsfeil som følge av dette, mens misoppfatninger er feil som oppstår av studentens manglende kunnskaper om programmering. Algoritmiske feil kan skyldes at studentene sliter med å omgjøre problemet til en matematisk form (Gomes, Carmo, Bigotte, & Mendes, 2006). Her er det viktig å poengtere at mens misoppfatninger kan relateres til manglende kunnskaper i programmering så trenger ikke algoritmiske feil eller feiltolkninger å gjøre det (Ettles, Luxton-Reilly, & Denny, 2018). I studien til Ettles, Luxton-Reilly og Denny (2018) ble det vist at studentene hadde færre feil knyttet til algoritmer og feiltolkning enn til misoppfattelser. Studentene brukte også mer tid på å rette opp feil som skyldtes misoppfattelser enn på feil i de to andre kategoriene.

2.2.5 Forkunnskaper i programmering

Studenter som tar høyere utdanning tar med seg en del varierende forkunnskaper inn i utdanningen. Forkunnskaper som har vist å påvirke studentenes evne til å programmere er blant annet problemløsningsferdigheter, matematikk og programmeringserfaring (Medeiros *et*

al., 2018). Studenter som har mangelfulle problemløsningsferdigheter kan ha utfordringer med å lære seg å programmering (de Lira Tavares *et al.*, 2012). Programmering er et komplekst fag å lære og utilstrekkelige problemløsningsferdigheter kan skape ekstra utfordringer for nye programmerere. Matematiske forkunnskaper kan også påvirke læring av programmering. Studenter som programmerer i høyere utdanning har ofte for dårlige forkunnskaper i matematikk til å programmere (Gomes & Mendes, 2010). Studenter som ikke har tilstrekkelige matematikkunnskaper kan slite med å lage en algoritme til problemet sitt. Enten klarer de ikke å lage algoritmen, eller så lager de den feil slik at algoritmen ikke virker som den skal. Sistnevnte kan da føre til algoritmiske feil (Ettles *et al.*, 2018). En tredje ting som kan påvirker programmering er hvorvidt de som programmerer har noen erfaring med programmering fra før (Medeiros *et al.*, 2018). Studenter med tidligere erfaringer viser ofte bedre ferdigheter og større selvtillit enn studenter som ikke erfaring (Hagan & Markham, 2000).

2.2.6 Å programmere i en faglig kontekst

Det som skiller BIOS1100 fra andre introduksjonskurs i programmering er at studentene programmerer i en biologisk kontekst. Kontekst i programmering kan defineres som all den relevante informasjonen en person trenger for å fullføre en oppgave (Chattopadhyay, Nelson, Nam, Calvert, & Sarma, 2018). Konteksten består ofte av forskjellige kilder med informasjon som programmereren må tolke, avhengig av hva målet med programmeringen er. I min studie ser jeg på programmering i en biologisk kontekst, hvor studentene må forholde seg til ulike former for biologisk informasjon.

I forskningslitteraturen er det noen studier som ser på effekten av å undervise programmering og biologi sammen. En studie gjort på dette feltet er gjennomført ved Harvey Mudd College. Her ble det utviklet et kurs for studenter i det første året, som skulle lære dem prinsippene bak CT og programmering gjennom biologiske problemstillinger (Dodds, Libeskind-Hadas, & Bush, 2012). En tilsvarende studie ble gjort senere ved Universitet i Illinois, og bygde på samme introduksjonskurs fra Harvey Mudd College (Berger-Wolf, Igic, Taylor, Sloan, & Poretsky, 2018). Begge studiene sammenliknet studenter i biologiske programmeringskurs, med studenter som enten tok rene introduksjonskurs i programmering (Dodds *et al.*, 2012) eller studenter som tok andre tilsvarende tverrfaglige introduksjonskurs i programmering (Berger-Wolf *et al.*, 2018). Resultater fra begge studiene viste at biologistudentene hadde i

gjennomsnitt like gode programmeringsferdigheter som de andre studentene. Dette gir en god indikasjon på at biologistudentenes evne til å programmere ikke ble påvirket av å måtte lære biologi og programmering i samme kurs. Samtidig viser studien til Dodds *et al.* (2012) at biologistudentene heller ikke brukte mer tid på å arbeide med kursinnholdet enn de andre studentene. Dette tyder på at biologistudentene ikke måtte bruke mer tid på å lære seg samme programmeringskunnskaper enn studentene i det rene programmeringskurset. Begge studiene viste også at biologistudentene generelt var positive til å programmere i biologi.

I Norge er det gjort få studier på programmering i biologi ved universitetene. Sørby og Angell (2012) så derimot på fysikkstudenter som arbeidet med programmering ved UiO. Sørby og Angell hadde laget programmeringsoppgaver som krevde at studentene brukte kunnskaper fra både fysikk, matematikk og programmering. Resultatene viste at studentene slet med å kombinere de ulike kunnskapene for å løse oppgavene. En annen studie som har sett på effekten av å programmere i kontekst er gjort av Bouvier *et al.* (2016). Deres studie viste at det å programmere i en kontekst ikke hjelper studentene med å løse problemer i programmering. Til tross for at studien deres ikke undersøkte det, argumentere forfatterne for at kontekstspesifikke oppgaver kan ha en positiv effekt på studentenes motivasjon for å løse oppgaver og å lære seg programmering. Dette samsvarer med to andre studier av studenter som programmerer i mer generelle kontekster (Berger-Wolf *et al.*, 2018; Forte & Guzdial, 2005). Samtidig er det andre utfordringer som kan knyttes til å programmere i en bestemt kontekst. Guzdial (2010) argumenterer for at svakere studenter i kontekstbaserte kurs kan bli distrahert av all informasjonen som inngår i konteksten. Studentene kan også slite med å overføre den kunnskapen de lærer fra en kontekst til en annen.

2.2.7 Forskning på gruppearbeid og programmering

Under gruppetimene i BIOS1100 arbeider studentene i grupper på 4-8 studenter og arbeider med å løse programmeringsoppgaver sammen. Bruk av gruppearbeid eller samarbeidende læring har vist å gi en positiv effekt på sluttvurderingen til studenter som tar introduksjonskurs i programmering (Beck, Chizhik, & McElroy, 2005). Det har også vist positiv effekt på kvaliteten på studentenes innleveringer (Urness, 2009). Samarbeid og diskusjoner har også vist positiv effekt på studenters læring av modeller i fysikk (Sørby & Angell, 2012)

I sin artikkel beskriver LeJeune (2003) fem faktorer som er kritiske for at samarbeidende læring skal fungere i introduksjonskurs:

1) Felles oppgave. Gruppa må arbeide med den samme oppgaven og at den krever et kollektivt bidrag fra gruppa.
2) Små grupper. Gruppestørrelsen bør være på rundt fem personer for å tillate direkte interaksjoner mellom studentene, samtidig må det være nok personer til å dele kunnskaper og meninger.
3) Samarbeidende adferd. Studentene må dele sine kunnskaper og ferdigheter med hverandre og være mottakelig for andres kunnskaper og ferdigheter.
4) Gjensidig avhengighet. Studentene i gruppa er ansvarlige for hverandres deltakelse og bidrag.
5) Individuelt og kollektivt ansvar. Hver student må bidra inn mot de andre i gruppa og de må ta ansvar for å oppsøke nye kunnskaper og ferdigheter fra de andre.

Studenten i BIOS1100 arbeidet sammen i grupper med en størrelse som oppfyller punkt 2) til LeJeune (2003). I utgangspunktet arbeidet studentene med de samme oppgavene, som oppfyller punkt 1), men mine observasjoner av gruppetimene indikerte at studentene stod fritt til å arbeide med de oppgavene de ønsker. Dermed er det ikke selvsagt at de jobbet med samme oppgaver. Hvorvidt de oppfyller de andre punktene kommer vi nærmere inn på i diskusjonen i kapittel 5.

2.2.8 Bruk av eksempelprogrammer og «samstemt undervisning»

BIOS1100 bruker livekoding som sin hovedform for undervisning. Livekoding kan brukes i flere ulike formater og i BIOS1100 var det lagt opp slik at en gruppelærer skrev programmer sammen med studentene og viste studentene hvordan de kunne løse spesifikke oppgaver. Flere studier viser at livekoding kan ha positiv effekt på studentenes ferdigheter i programmering (Rubin, 2013; Shannon & Summet, 2015). Slik livekodingen ble benyttet i BIOS1100 så tok de i bruk eksempler for å lære studentene å skrive programmer. I følge Neal (1989) kan eksempler ha en rekke funksjoner under læring av programmering. Det kan brukes for å vise hvordan programstrukturer virker, hvordan algoritmer skal settes inn i programmer eller for å vise hvordan problemer kan løses. Bruk av eksempler er en prosess som også erfarne programmere benytter ved å gjenbruke hele eller deler av andre personers programmer (Neal, 1989). Bruk av eksempler spiller derfor en viktig rolle hos både nybegynnere og

erfarne programmerere. En studie av Müller, Silveira, og Souza (2018) har undersøkt hvordan og hvorfor nybegynnere bruker eksempleprogrammer når de programmerer. Studien viste at studentene hovedsakelig bruker eksempler for å 1) forstå programmeringsspråket, 2) forstå problemet, 3) optimalisere programmet og 4) hjelpe dem om de står fast i problemet. Videre så de at studentene brukte eksempelprogrammer enten ved at de brukte dem som en referanse, eller ved at de kopierte eksempelprogrammet. Å bruke eksempelprogrammene som referanse betydde at studentene brukte dem for å se hvordan de kunne lage sine egne programmer. Når de kopierte eksempelprogrammene kopierte de enten hele, eller deler av programmet for å løse tilsvarende oppgaver. I relasjon til eksempelbasert læring har Gaspar og Langevin (2007) beskrevet det de definerer som «klipp og lim» orientert programmering. Med dette mener de at når studenter møter nye oppgaver så starter de med å identifisere nøkkelord som kan relateres til tidligere oppgaver. Gaspar og Langevin (2007) er skeptiske til denne formen for programmering og stiller spørsmåltegn ved hvorvidt dette fostrer læring og motivasjon hos studentene.

Ett av målene i BIOS1100 er at studentene skal ha fortrolighet med programmering i Python (Universitetet i Oslo, 2018). For at studentene skal nå læringsmålene som er satt for dem, må læringsaktivitetene være tilpasset læringsmålene (Biggs & Tang, 2011). For å få til dette har Biggs og Tang (2011) definert begrepet «Constructive alignment», som handler om at det må være sammenheng mellom læringsmålene, lærings- og undervisningsformene, arbeidskravene og sluttvurderingen. Oversatt kan vi kalle «Constructive alignment» for samstemt undervisning. Biggs og Tang har laget en liste med tiltak som kan gjennomføres for å sikre samstemt undervisning. Blant disse tiltakene er: 1) beskrive læringsutbytte sånn at studentene forstår hva som kreves av dem, 2) utforme undervisning som gir studentene mulighet til å oppnå læringsmålene, 3) bruke vurderingsformer som gjør det mulig å bedømme studentenes prestasjon og 4) bruke vurderingen til å lage standard vurderingskriterier. I BIOS1100 har studentene tilgang på læringsmål for emnet og gjennom livekoding og gruppearbeid blir studentene presentert for hvordan programmer skal skrives. Dermed kan man si at BIOS1100 oppfyller punkt 1) og 2) for samstemt undervisning. Samtidig er eksamen i BIOS1100 en individuell eksamen der studentene ikke får muligheten til å kjøre programmene de lager. Det er derfor mer usikkert om læringsaktivitetene samsvarer med vurderingsformen i BIOS1100, og dermed om punkt 3) for samstemt undervisning blir møtt.

3 Metode og analyse

Denne studien har som hensikt å undersøke hvordan studenter ved UiO arbeider med programmeringsoppgaver i biologi. Informantene i studien var både menn og kvinner som deltok i BIOS1100. Det ble gjort lydopptak av tre grupper som løste ett sett med programmeringsoppgaver i biologi og deretter deltok i et fokusgruppeintervju. Resultatene i denne studien bygger også på et spørsmål fra et spørreskjema som ble gjennomført i slutten av semesteret på BIOS1100 og som ble delt ut til alle studentene på emnet.

I dette kapittelet vil jeg starte med å begrunne valg av forskningsdesign, metode og informanter. Videre vil jeg redegjøre for utforming og gjennomføring av programmeringsoppgaver, fokusgruppeintervjuer og spørreskjema. Deretter vil jeg drøfte analysen av dataene og valgene som er blitt tatt underveis. Til slutt vil jeg diskutere reliabiliteten og validiteten til denne studien.

3.1 Valg av forskningsdesign og metode:

Denne studien tar utgangspunkt i et kvalitativt forskningsdesign. Dalen (2011) beskriver målet med kvalitativ metode som følgende:

«Et overordnet mål for kvalitativ forskning er å utvikle forståelse av fenomener som er knyttet til personer og situasjoner i deres sosiale virkelighet» (Dalen, 2011, s.15).

Når hensikten med studien er å undersøke hvordan studenter arbeider med programmeringsoppgaver, vil et kvalitativt forskningsdesign bidra til å utvikle forståelse av hvordan studentene arbeider og dermed være best egnet til å svare på problemstillingen. Samtidig vil det gi et mer nøyaktig bilde av studentenes arbeidsvaner om dette undersøkes i en autentisk arbeidssituasjon for studentene.

Intervju er godt egnet for å få innsikt i informanters tanker, følelser og erfaringer (Dalen, 2011). Når man bruker intervju som metode er man ikke ute etter å tallfeste meninger, men heller utforske bredden av et tema (Bauer & Gaskell, 2000). For denne studien har jeg valgt å bruke fokusgruppeintervjuer som metode fremfor enkeltintervjuer. Hensikten med fokusgruppeintervju er å få respondentene til å prate og interagere med hverandre. Fokusgruppeintervjuer kan bidra til å få frem flere spontane ekspressive og emosjonelle

synspunkter og kan derfor gi et mer genuint bilde av de sosiale interaksjonene, enn hva man får i enkeltintervjuer (Bauer & Gaskell, 2000). Med min studie har jeg ønsket å undersøke hvordan studentene arbeider med programmeringsoppgaver. Da var det ønskelig å observere studentene i en så autentisk situasjon som mulig. Siden studentene på BIOS1100 arbeidet i faste grupper i gruppetimene, ble det naturlig å gjennomføre intervju med gruppene studentene arbeidet i til vanlig. Et annet hensyn som jeg måtte ta var omfanget av oppgaven min. Ved å bruke fokusgrupper fikk jeg dekket meningene til langt flere respondenter enn jeg ville klart ved å bruke intervjuer av enkeltpersoner.

Samtidig er det noen utfordringer med fokusgruppeintervjuer. Cohen *et al.* (2011) påpeker at fokusgrupper ofte kan preges av samtaler der enkelte respondenter ikke deltar, mens andre dominerer. Dette kan resultere i at dataene fra fokusgruppen mangler en overordnet reliabilitet, der funnene ikke samsvarer med gruppens helhetlige synspunkter. Det kan også være utfordrende å få alle informantene til å møte opp samtidig (Bauer & Gaskell, 2000). Jeg løste derimot dette veldig greit ved å intervjue gruppene etter de var ferdig med livekodingen i gruppetimen. Terskelen for å møte opp ble dermed mindre for studentene, siden de allerede var tilstede i gruppetimen.

Jeg har også valgt å kombinere fokusgruppeintervjuene med en observasjon av gruppene mens de løser programmeringsoppgaver. Måten jeg gjennomførte dette på var at gruppene først arbeidet med oppgavene, for så å bli intervjuet etterpå. Ved å kombinere en oppgave med fokusgruppeintervju kunne jeg dermed få et innblikk i både hvordan studentene arbeider, men også hva de selv tenker og mener rundt arbeidet sitt. Samtidig fikk oppgavene belyst hvordan studentene arbeidet med programmeringsoppgaver i en så autentisk situasjon som mulig.

3.2 Oppgaveløsing og fokusgruppeintervju

3.2.1 Utforming av programmeringsoppgavene

For å undersøke hvordan studentene programmerer i biologi ville vi bruke oppgaver som krevde at studentene måtte anvende kunnskaper om både programmering og biologi samtidig. En forutsetning for oppgavene var at de var på et nivå der studentene hadde tilstrekkelige kunnskaper innen både programmering og biologi til å løse dem. Dermed måtte det omhandle

temaer som studentene allerede hadde hatt undervisning i på UiO. Prosessen med å velge oppgaver ble gjort i samråd med mine tre veiledere. En av veilederne mine var emneansvarlig i BIOS1100 og hadde oversikt over hvilke typer oppgaver studentene arbeidet med og hva som var undervist fra før. Min andre veileder var emneansvarlig for «Celle og molekylærbiologi» (BIOS1110), et annet emne som ble undervist i første semester for studenter som studerte Biovitenskap ved UiO. For studentene som enten gikk Bachelor i Biovitenskap, Lektorprogrammet i Biologi og Kjemi/naturfag eller Årsenhet i Biologi, var det kun BIOS1100 og BIOS1110 som omhandlet biologi dette semesteret. Læringsmålene for disse emnene var derfor et godt utgangspunkt for hva studentene skulle kunne om både programmering og biologi.

I samarbeid med veiledere ble det valgt å bruke et oppgavesett (se vedlegg C) som det allerede var ment at studentene skulle arbeide med i en av gruppetimene. Fordelen med å bruke en allerede eksisterende oppgave var at den var på et format studentene var kjent med. Dette bidro til at studentenes arbeid med oppgaven ble mest mulig autentisk i forhold til hva de var vant med. Oppgavene var også laget med utgangspunkt i temaer som studentene nylig hadde lært om i både BIOS1100 og BIOS1110. Studentene skulle derfor ha de nødvendige forutsetningene for å kunne løse oppgavene. Tema i oppgavene var genetikk og arv. Studentene måtte bruke kunnskaper om arv til å lage et krysningsskjema, regne på sannsynlighet og lage et program som kunne regne ut genotypen og fenotypen til barn av foreldre med en heterozygot genotype (se vedlegg D for løsningsforslag). Det var i utgangspunktet fire spørsmål (oppgave a-d) i det oppgavesettet, men etter gjennomføring med første gruppe valgte jeg å kutte det ned til tre spørsmål (oppgave a-c), da jeg følte tre spørsmål var tilstrekkelig for å svare på min problemstilling.

3.2.2 Intervjuguide

Når man bruker intervju som forskningsdesign er det behov for å lage en intervjuguide (Dalen, 2011). Intervjuguiden (vedlegg A) ble laget i samarbeid med mine veiledere og bestod av fire hovedspørsmål og flere underspørsmål til å utdype hver av dem. Spørsmålene var laget med utgangspunkt i denne studiens problemstilling og var ment å avdekke studentenes tanker om oppgavene de arbeidet med før intervjuet, utfordringer, strategier, og motivasjon. Det første spørsmålet var ment som en myk åpning for fokusgruppene, der intensjonen var å få dem til å føle seg komfortable i intervjusituasjonen. Intervjuguiden la opp

til at jeg ville stille hovedspørsmålene etter hverandre, mens underspørsmålene ble spurt avhengig av samtalen til studentene. Et intervju som er på denne formen kalles *semistrukturert* (Kvale & Brinkmann, 2015). Semistrukturerte intervjuer kan ha ulike grader av rigiditet i sin form og det er studien som bestemmer hvor rigid strukturen er. I min intervjuguide valgte vi å ha en nokså rigid struktur siden jeg ønsket å dekke alle spørsmålene vi hadde forberedt. For å ha kontroll over spørsmålene markerte jeg hvert spørsmål etter jeg hadde fått svar på det. I de tilfellene der studentene svarte på et spørsmål uten at jeg stilte det, ble det markert og jeg lot være å stille studentene det spørsmålet. Dette bidro til en mer dynamisk samtale der studentene slapp å gjenta svarene sine flere ganger.

Spørsmålene i et intervju skal være korte og forståelige (Kvale & Brinkmann, 2015). Ved utforming av intervjuguiden forsøkte vi derfor å lage spørsmål som var konkrete og uten tunge faglige begreper. Vi valgte også å lage spørsmål som først og fremst var åpne. Åpne spørsmål legger i liten grad føring på hva svaret blir (Robson, 2002). Fordelen med åpne spørsmål er at de kan gi innsikt i hva informantene faktisk mener og at man kan gå i dybden av det man spør om. For min studie var derfor åpne spørsmål et bedre alternativ enn lukkede, siden jeg ønsket å få innsyn i hvordan studentene arbeidet med programmeringsoppgaver og hvilke meninger de hadde om programmering i biovitenskapelige problemstillinger.

3.2.3 Utvalg og rekruttering av informanter:

Ettersom formålet med studien er å undersøke hvordan studenter arbeider med programmeringsoppgaver i biologi, valgte vi å bruke studenter på BIOS1100 som informanter. Informantene ble rekruttert gjennom gruppetimene i emnet. En uke før intervjuet valgte jeg å observere tre forskjellige gruppetimer for å skaffe informanter til undersøkelsen. Firebaugh (2008) påpeker at når man bruker et utvalg for å representere en populasjon, er utvalgets evne til å representere populasjonene viktigere enn størrelsen på selve utvalget. Jeg valgte å rekruttere studenter fra tre forskjellige gruppetimer siden de ble undervist av forskjellige gruppelærere. Grunnet individuelle forskjeller mellom gruppelærerne vil undervisningen være litt forskjellig i de tre gruppetimene. Dette kan ha medført at studentene hadde forskjellige erfaringer og kunnskaper om programmering og biologi, så ved å rekruttere fra de ulike gruppetimene kunne jeg få et bredest mulig utvalg av studenter. Det som ofte kan være en enkel metode for å skaffe et representativt utvalg er å tilfeldig velge individer fra en populasjon (Firebaugh, 2008). Samtidig påpeker Firebaugh at det ofte kan være utfordrende å

få til nettopp dette. Fordelen med å velge allerede etablerte grupper fra gruppetimene var at disse gruppene ble tilfeldig satt sammen ved kursstart av de som hadde ansvar for BIOS1100. Dermed rekrutterte jeg grupper hvor studentene var tilfeldig satt sammen, noe som bidrar til et mer representativt utvalg for studentene ved BIOS1100.

Det å bruke en allerede etablert gruppe bidro til autentisiteten av undersøkelsen. Studentene hadde jobbet i de samme gruppene i to måneder og ved å observere dem i den samme gruppa var det en situasjon som var naturlig for dem. Ved valg av grupper hadde jeg på forhånd satt to kriterier for rekruttering. Gruppene skulle bestå av et noenlunde likt antall menn og kvinner og det skulle være 4-8 personer på gruppa. Videre valgte jeg å henvende meg til de gruppene om viste stor grad av dialog i gruppa. Dette gjorde jeg for å rekruttere grupper som var trygge på hverandre og som så ut til å prate godt sammen. Spesielt siden jeg skulle ta lydopptak av arbeidet deres, trengte jeg grupper som pratet mye sammen.

Studentene ble rekruttert etter at livekodingen var ferdig i gruppetimen. Det ble gjort ved at jeg oppsøkte gruppene jeg ønsket å rekruttere. Gruppene ble informert om at jeg var en lektorstudent som skulle skrive en masteroppgave om hvordan studenter programmerte i biologi. Videre informerte jeg at jeg ønsket å gjøre et lydopptak av dem mens de gjorde noen oppgaver i programmering, etterfulgt av et fokusgruppeintervju. Gjennomføringen av oppgaveløsning og fokusgruppeintervjuet skulle gjøres etter livekodingen i den neste gruppetimen de hadde. Studentene ble informert om at de ville forbli anonyme, og at de fikk pizza som takk for deltakelsen. I den første gruppetimen jeg prøvde å rekruttere fra takket den første gruppen nei. Den andre gruppa jeg forsøkte å rekruttere bestod av fire studenter og de takket ja. Når jeg rekrutterte fra den andre og tredje gruppetimen sa de første gruppene ja med en gang. Begge disse gruppene bestod av fem studenter. Ved gjennomføring av oppgaveløsning og fokusgruppeintervju hadde jeg derimot noen studenter som ikke dukket opp. Endelig antall studenter og fordeling av kjønn ved gjennomføringen av datainnsamlingen er oppgitt i tabell 2.

Tabell 2 – fordeling av kjønn og antall studenter per gruppe under datainnsamling

Gruppe	**Antall studenter per gruppe ved rekruttering**	**Antall studenter per gruppe ved datainnsamling**	**Fordeling av kjønn per gruppe ved datainnsamling**
1	4	2	2 kvinner
2	5	5	3 kvinner 2 menn
3	5	4	3 kvinner 1 mann

3.2.4 Gjennomføring av gruppeoppgaver og fokusgruppeintervju

Fokusgruppeintervjuene ble gjennomført i slutten av oktober og datainnsamlingen av alle tre gruppene ble gjort i løpet av en uke. Datainnsamlingen ble gjort sent i semesteret for å sikre at studentene hadde best forutsetninger og kunnskaper til å gjennomføre oppgavesettet. Samtidig måtte vi ta hensyn til studentenes eksamen i BIOS1100, og dermed ble det ikke gjennomført senere enn oktober. Datainnsamlingen ble gjort i samme tid som studentene hadde gruppetimer i BIOS1100. Når livekodingen var ferdig ble gruppene tatt med til et naborom der datainnsamlingen ble gjennomført. Studentene i hver gruppe fikk lese et samtykkeskjema (vedlegg B) og skrev under på en samtykkeerklæring til studien. Deretter informerte jeg studentene om hensikten med studien, hva lydopptakene skulle brukes til og at studentene når som helst og uten begrunnelse kunne trekke seg (vedlegg A). Studentene ble oppfordret til å diskutere mest mulig med hverandre når de løste oppgaven. De ble også informert om at de hadde tilgang på en hjelpelærer ved behov. Dermed ble situasjonen mest mulig lik den de vanligvis var i når de arbeidet med oppgaver i gruppetimene. Studentene fikk delt ut oppgavene på et ark og hadde tilgang på en større dataskjerm hvor de kunne koble opp en datamaskin. Et generelt trekk for intervjuene var at alle gruppene var preget av god dialog. De fleste kom med innspill under store deler av intervjuet og studentene virker trygge på hverandre. Samtidig stoppet samtalen opp en del og jeg som moderator ble tvunget til å stille oppfølgingsspørsmål for å drive samtalen frem. Under intervjuene forsøket jeg å vente lengst mulig før jeg stilte nye spørsmål, for å ikke avbryte eventuelle ekstra tanker studentene ønsket å dele.

Gruppe 1:

I den første gruppen var det kun to studenter som møtte opp. De brukte rundt én time på gjennomføring av oppgavene, mens intervjuet tok rundt 30 minutter. Som nevnt i avsnitt 3.2.1 var det i utgangspunktet fire spørsmål i oppgavesettet. Jeg lot studentene bruke den tiden de trengte til å gjøre oppgave a-c i oppgavesettet, men etter at studentene hadde arbeidet i 10 minutter med oppgave d valgte jeg å stoppe dem. Grunnen til det var at jeg følte jeg hadde fått nok informasjon fra de tre første oppgavene og at vi trengte tid til intervjuet. De to studentene i gruppe 1 snakket mye sammen, både under oppgaveløsningen og ved intervjuet. En gang måtte de spørre om hjelp. Det var for å få bekreftet genotypene til besteforeldre og foreldre i oppgave a. Intervjuet ble avbrutt en gang, da det var en telefon som ringte.

Gruppe 2

Fra gruppe 2 deltok alle fem studentene. De brukte rundt 30 minutter på å løse oppgaven og studentene løste alle oppgavene på egenhånd. Da jeg påpekte at de hadde svart feil på oppgave b, endte det opp med en diskusjon rundt denne oppgaven som varte i 40 minutter. Intervjuet varte i 50 minutter. Som følge av levering av to forskjellige pizzaer ble dette intervjuet avbrutt to ganger, noe som kan ha påvirket flyten i intervjuet. Det må også påpekes at intervjuet ikke var ferdig før kl. 19.00, så det kan tenkes at studentene begynte å bli slitne og lei mot slutten. Dermed et det en mulighet for at svarene som ble gitt mot slutten er preget av at studentene ønsket å bli fort ferdig med intervjuet.

Gruppe 3

Fra den siste gruppen møtte fire av fem studenter og de brukte rundt 30 minutter på å løse oppgavene og deretter rundt 30 minutter på intervjuet. Pizzaen ble levert i tiden mellom de løste oppgavene og intervjuet, så det ble ingen avbrytelser her. Heller ikke denne gruppen spurte om hjelp til oppgavene. Denne gruppen ble ferdig med intervjuet 18.30, så også her kan vi ha møtt utfordringen med at studentene var slitne mot slutten av intervjuet og ønsket å bli fort ferdig.

3.3 Spørreskjema – Utforming og gjennomføring

Spørreskjemaet ble utført i samarbeid med masterstudenten Marthe Mjøen Berg og mine og hennes veiledere. Min masteroppgave har kun tatt utgangspunkt i ett av spørsmålene i spørreskjemaet og jeg har kun brukt dette spørsmålet for å underbygge resultatene fra mine egne fokusgruppeintervju og observasjoner. Jeg velger derfor å ikke beskrive dette spørreskjemaet i stor detalj her, men henviser heller til Marthes studie for hvordan denne ble laget (Berg, 2019). Siden jeg kun har brukt ett spørsmål har jeg valgt å ikke legge med spørreskjemaet som et vedlegg. Spørsmålet jeg har brukt i min studie er:

Hva har du opplevd som mest utfordrende når du har programmert og modellert biologiske problemstillinger i Python?

Formålet med spørreskjemaet var å kartlegge hva som kjennetegner studentene på BIOS1100 og hvordan studentenes interesseverdi og mestringsforventning i BIOS1100 endret seg gjennom semesteret. Interesseverdi er en verdi som bidrar til å forklare prestasjonsrelaterte valg, gjennom den verdien en person legger i en aktivitet. Interesseverdi viser til glede og interesse for aktiviteten. Spørreskjemaet skulle sammenlikne studentenes interesseverdi og mestringsforventning i BIOS1100 mot resten av studieprogrammet de gikk på. Spørreskjemaet ble gitt ut til studentene ved BIOS1100 ved to anledninger. Den første spørreundersøkelsen ble gjort under studentenes første forelesning i august, mens den andre ble gjort i dere siste forelesning i november. Gjennomføringen av spørreundersøkelsen ble gjort ved å bruke nettsiden Nettskjema.uio.no. Spørsmålene ble forsøkt å være så like som mulig, men det ble gjort noen endringer underveis. Spørsmålet jeg har brukt i min analyse er kun fra det siste spørreskjemaet. Ved gjennomføring av den siste spørreundersøkelsen var det 103 studenter som svarte mot 149 som svarte ved den første gjennomføringen i august. Dette henger nok sammen med det at det første spørreskjemaet ble levert ut i en obligatorisk forelesning, mens det var frivillig oppmøte i den forelesningen der siste spørreskjema ble levert ut. Av de 103 personene som svarte på spørreskjemaet var det kun 77 personer som svarte på spørsmålet jeg har analysert.

3.4 Analysemetode

3.4.1 Transkribering

Transkripsjon handler om å gjøre tale om til tekst. Under fokusgruppeintervjuene og gruppeoppgavene valgte jeg å bruke en lydopptaker til å ta opp hva studenten snakket om. Fordelen med dette er at jeg ikke bare fikk med meg hva studentene snakket om, men også tonefall, pauser, latter og slike ting. Sånne observasjoner er viktig å ha med i transkriberingen, siden et intervju er en levende, sosial interaksjon med mer enn bare ord (Kvale, 2007) Samtidig gav dette meg muligheten til å fokusere på selve intervjuet og dermed stille gode, oppfølgende spørsmål underveis. Jeg valgte å skrive ned observasjoner jeg anså som viktige, noe som har bidratt til at jeg har kunne gi tydeligere beskrivelse av gruppene i dette avsnittet.

Transkriberingen bør starte så raskt som mulig etter at fokusgruppeintervjuet er gjennomført (Krueger & Casey, 2002) Min datainnsamling skjedde i samme periode som jeg hadde egne eksamener og derfor ble ikke transkribering gjort før en måned etter intervjuene. Ulempen med å vente så lenge er at det ble vanskeligere for meg å huske situasjonene hvor lydopptaket foregikk. Samtidig har jeg underveis i lydopptakene tatt gode notater, noe som gjorde det lettere for meg å forstå situasjonen jeg transkriberte. Siden det var jeg som gjennomførte intervjuene og transkribering, har jeg også fått god innsikt i dataene. Fordelen med det er at jeg får bedre kjennskap til datamaterialet og at jeg allerede under transkriberingen kan begynne analysen (Kvale, 2007). Dette kommer jeg nærmere innpå i avsnittet om den tematiske analysen.

Når man transkriberer bør man forsøke å skrive så ordrett som mulig (Krueger & Casey, 2002). Informantene vil ikke alltid gi fulle setninger og da er det viktig at man ikke fyller inn setningene deres. Under transkriberingen har jeg derfor forsøkt å skrive av så ordrett som mulig. Jeg valgte å skrive på bokmål, men uttrykk som ikke hadde noen direkte oversettelse til bokmål, ble skrevet som de ble uttalt. Jeg har også valgt å skrive ned studentens handlinger som (latter), eller (pause), og hvis jeg ikke fikk med meg hva studentene sa i en setning, ble dette transkribert som (…). For å sikre at jeg transkriberte så korrekt som mulig, sjekket jeg det jeg hadde skrevet mot lydopptakene to ganger.

3.4.2 Tematisk analyse

Jeg har i denne studien valgt å bruke tematisk analyse av mine data. Tematisk analyse er en metode for å identifisere, analysere og rapportere temaer i dataene du har (Braun & Clarke, 2006). Denne metoden handler om å analysere temaer over flere datasett, fremfor å se etter temaer innenfor hvert datasett. Fordelen med tematisk analyse er at den er veldig fleksibel og ikke er bundet til en bestemt form for teoretisk rammeverk i steg selv. Når man arbeider med en så fleksibel analyse er det derfor viktig å gi en god avklaring av hvilke valg man tar under analysen (Braun & Clarke, 2006). I dette avsnittet vil jeg derfor forsøk å gi en så grundig forklaring på min analyse at det er ikke er tvil om hvordan den har blitt utført.

Det teoretiske rammeverket til en studie tar med seg en rekke antakelser om dataene man analyserer (Braun & Clarke, 2006). Siden jeg brukte CT som rammeverk for min studie, har jeg valgt å analysere forskningsspørsmålene mine under de seks komponentene som utgjør modellen til Shute *et al.* (2017). Dette valget innebærer at jeg i større grad kan beskrive hvordan studentene arbeider gjennom CT, men det kan også føre til at interessante resultater som faller utenfor komponentene til Shute *at al*. ikke kommer med i analysen. En annen ting som er viktig å poengtere med tematisk analyse er at det ikke er noen skjulte temaer som plutselig kommer frem i dataene (Braun & Clarke, 2006). Forskeren spiller en aktiv rolle i å finne temaer i dataene og det er forskeren som velger hva som er interessant å undersøke. Man kan ikke som forsker fri seg helt fra de teoretiske eller epistemologiske forankringene til studien.

Det er viktig å konstatere at tematisk analyse handler om å ta flere valg både før og under analyseprosessen (Braun & Clarke, 2006). Når jeg har jobbet med analysen har både CT modellen til Shute *et al.* (2017) og forskningsspørsmålene dannet grunnlaget for hva jeg har sett etter i dataene. Når analysen preges av enten det teoretiske rammeverket eller forskningsspørsmålene kalles det for en deduktiv analyse (Braun & Clarke, 2006). Utfordringen med å bruke en deduktiv analyse er ifølge Braun & Clarke at den tematiske analysen gir en mindre rik beskrivelse av dataene, noe som kan føre til at interessante observasjoner forsvinner. Til tross for en deduktiv tilnærming, har jeg forsøkt å være åpen for informasjonen i datasettene. Forskningsspørsmålene har ligget som en overordnet del under analysen, men kodene som oppstod ble ikke kategorisert inn i de seks komponentene av CT før på slutten av analysen. Når jeg gjorde dette endte jeg opp med å legge inn et ekstra

forskningsspørsmål, som er det som omhandler studentenes arbeidsvaner med programmering.

Braun og Clarke (2006) har definert seks faser av hvordan tematisk analyse kan gjennomføres. Jeg har arbeidet etter disse seks fasene og vil nå gi en beskrivelse av dem og mitt arbeid:

1. Forskeren må først gjøre seg kjent med dataene. I mitt arbeid transkriberte jeg først dataene, og tok notater av sitater som virker interessante. Når transkriberingen var ferdig, leste jeg gjennom den transkriberte teksten to ganger og skrev ned det som virket interessant. Svarene på spørsmålet fra spørreskjemaet og de transkriberte tekstene ble lastet inn i analyseprogrammet Atlas.ti (ATLAS.ti GmbH).

2. Med utgangspunkt i dataene, lager forskeren de første kodene. Jeg startet med å notere ned noen generelle koder med utgangspunkt i mine notater fra fase 1. Jeg laget koder direkte inn i Atlas.ti og startet med å analysere svarene fra spørreskjemaet. Dette gav meg et utgangspunkt for hvilke utfordringer studentene hadde i emnet, som jeg kunne bygge videre på i analysen. Deretter fortsatte jeg å skrive inn koder i de resterende dokumentene. Først jobbet jeg med fokusgruppeintervjuene, deretter med gruppeoppgavene. Mens jeg kodet så jeg etter funn som kunne relateres til studentenes utfordringer, strategier, holdninger og arbeidsvaner. Med Atlas.ti kunne jeg derfor bruke de samme kodene over de ulike tekstfilene og kunne sammenlikne om kodene jeg lagde faktisk samsvarte med hva studentene fortalte.

3. Når kodene er laget skal de kategoriseres inn i ulike temaer. Jeg ønsket å undersøke likheter og ulikheter mellom de tre gruppene jeg snakket med. For å ha mest kontroll over kodene, valgte jeg å dele opp denne fasen i tre deler. I den første fasen undersøkte jeg kodene mellom fokusgruppeintervjuene. Med den andre fasen undersøkte jeg kodene mellom gruppenes arbeid med programmeringsoppgaver, og i den siste så jeg på kodene fra spørreskjemaet. Under dette arbeidet valgte jeg å plassere kodene inn i komponentene av CT (Shute *et al.*, 2017). Komponentene til Shute *et al.* ble derfor definert som temaer i analysen. I tillegg oppstod det også tre

andre temaer som var *eksempelprogrammer, biologi og programmering, og arbeidsvaner*

4. Når kodene er blitt satt sammen til temaer, må de revideres og finpusses. På dette nivået skal dataene fra fase 1 sammenliknes med dem fra fase 2, og temaene skal revideres og korrigeres. For min analyse involverte dette arbeidet å se over kodene i hvert tema jeg hadde laget og se om innholdet i hvert tema representerte temaet på en god måte. Noen koder ble fjernet, noen ble slått sammen til én kode og nye koder ble lagt inn. Jeg gjentok den sammen prosessen med temaene. Dette resulterte i at to av temaene, som er Iterasjon og Generalisering i modellen til Shute *et al.* (2017) ble fjernet, ettersom dataene ikke var tilstrekkelig til å si noe om dette. Arbeidet ble gjennomført i Atlas.ti hvor jeg hadde muligheten til å gå inn i en og en kode og sammenlikne de ulike sitatene mot hele intervjuet koden var en del av.

5. Den femte fasen handler om å definere og navngi temaene. For min del var dette forhåndsbestemt av modellen til Shute *et al.* (2017). Mitt arbeid bestod derfor i stor grad av å definere kodene som var plassert i hvert tema. I avnsitt 4.2 definerer jeg hver kode slik at leseren får innsikt i hva som utgjør hver av dem.

6. Den siste fasen av tematisk analyse er å produsere rapporten. Denne fasen har da blitt det som er metode- og resultatkapittelet i min oppgave.

Tematisk analyse er ikke en lineær prosess der man går gjennom de ulike fasene og avslutter med fase seks (Braun & Clarke, 2006). I stedet arbeider man seg frem og tilbake mellom de ulike fasene, revurderer og omskriver. For min del betyr det at jeg har arbeidet meg frem og tilbake i de ulike dataene. Jeg har gått tilbake og lyttet til lydopptakene når jeg har vært usikker, og jeg har sammenliknet de ulike intervjuene mot hverandre.

3.5 Troverdighet

3.5.1 Reliabilitet

Reliabilitet betyr at studien er troverdig og at resultatene kan reproduseres fra andre, liknende studier (Cohen, Morrison, & Manion, 2011). For at en studie skal ha reliabilitet, må man kunne forvente å få tilsvarende resultater om man gjennomfører studien i en liknende

situasjon. Cohen *et al.* (2011) påpeker samtidig at det ikke er like lett å drøfte reliabilitet i kvalitative studier, ettersom resultatene er et produkt av den unike situasjonen undersøkelsen er gjort i. Allikevel bør det forsøkes å arbeide for at resultatene kan reproduseres. For at andre skal kunne repetere min studie er det viktig at informasjon om hvordan studien er gjennomført er tydelig redegjort for. Jeg har forsøkt å gjøre det i dette kapittelet ved å presentere intervjueguiden (vedlegg A), valg av fokusgrupper, gjennomføring av intervju, gruppenes arbeid med oppgavene og hvordan jeg har analysert dataene.

Reliabiliteten kan også styrkes under både intervju og transkribering. Når man transkriberer bør man lytte til opptaket flere ganger, for å avdekke avvik som feiltolkning av hva informantene sier eller hvordan setninger er bygget opp (Kvale, 2007). Etter at jeg hadde fullført transkriberingen, sjekket jeg dokumentene opp mot lydopptakene to ganger. Når man avdekker feil i transkripsjonen bidrar man til at den transkriberte teksten samsvarer med informantenes meninger og dermed styrker man reliabiliteten til studien.

En måte å sikre reliabilitet i kvalitative studier er å bruke en observatør med samme teoretiske rammeverk til å tolke de samme resultatene (Denzin & Lincoln, 1994). Under analyseprosessen ble det satt av to timer der jeg analyserte et utdrag av resultatene sammen med to av mine veiledere. Gjennom dette arbeidet fikk jeg bekreftet at mine tolkninger av resultatene samsvarte med tolkningene til mine veiledere og vi fikk drøftet betydningen av resultatene sammen.

3.5.2 Validitet

Validitet i kvalitative studier handler om hvor korrekte og sannferdige tolkningene er i forskningen (Johnson & Christensen, 2013). En mulig trussel mot validiteten er det vi kaller for *forskerbias*. I kvalitative studier er det forskeren som aktivt analyserer og fortolker dataene. Forskerbias oppstår ofte fordi forskerens tolkning blir påvirket av forskerens egne personlige synspunkter og perspektiver (Johnson & Christensen, 2013). En måte å forebygge forskerbias på i studien er gjennom *refleksivitet*, som betyr at man som forsker er bevisst egne synspunkter og hvordan de kan påvirke forskerens studie. Jeg vil argumentere for at en av mine fordeler med denne studien er at jeg ikke har mer erfaring med programmering enn et 10 studiepoengs emnet fra første semester i lektorutdanning. Dermed går jeg inn i studien med mindre synspunkter på programmering enn hva jeg ville hatt med mer erfaring. Samtidig var jeg bevisst på at jeg har hatt noen forventinger, som at programmering vil oppleves som mer

utfordrende for studentene enn biologi. Derfor prøvde jeg å ikke la disse forventningene påvirke hvordan jeg analyserte og fortolket dataene.

Cohen *et al.* (2011) deler validitet inn i ekstern og intern validitet. Intern validitet handler om at fortolkningen av datasettet kan støttes av dataene. En metode jeg har benyttet meg av i min studie er *triangulering*. Dette henviser til en teknikk der man anvender forskjellige forskere, metoder, datasett og/eller teoretiske perspektiver for å styrke validiteten til forskningen (Johnson & Christensen, 2013). Jeg har i denne studien brukt både fokusgruppeintervju, observasjon av arbeid med oppgaver og spørreundersøkelse for å svare på mine forskningsspørsmål. De ulike datasettene har blitt sammenliknet med hverandre for å styrke, eller stille spørsmålstegn til de tolkningene jeg har gjort. Samtidig har jeg også kodet deler av datasettet sammen med mine veiledere, og dermed tatt i bruk flere forskere for å undersøke samme datasett. Med utgangspunkt i egen refleksivitet og triangulering mener jeg at resultatene jeg presenterer i kapittel 4 og 5 representerer datasettene jeg har brukt på en god måte.

Ekstern validitet handler om hvorvidt resultatene fra studien kan generaliseres til å gjelde for andre enn dem som har deltatt i studien (Cohen *et al.*, 2011). Normalt pleier man ikke å generalisere resultatene fra en kvalitativ studie. Som regel forsøker man heller å gi en rik beskrivelse av en bestemt gruppe mennesker (Johnson & Christensen, 2013). Hensikten med min studie har først og fremst vært å beskrive hvordan studentene ved BIOS1100 arbeider med programmeringsoppgaver i biologi. Gitt at mine informanter i intervjuet var plukket fra tre forskjellige grupper, så vil dette gi gode forutsetninger for å representere studentene ved BIOS1100. Samtidig har jeg også brukt spørreskjemaet til å underbygge hvilke utfordringer studentene har møtt. Jeg kan derimot ikke generalisere mine funn til å gjelde andre studenter i liknende programmeringskurs, men jeg mener at mine funn viser eksempler på hvordan studentene arbeider med programmering i biologi.

4 Resultater

Denne studien undersøker hvordan studenter løser biovitenskapelige problemstillinger gjennom programmering. Innunder dette spørsmålet ønsker jeg konkret å se hvilke utfordringer og strategier studentene har når de programmerer, hvordan biovitenskapelige problemstillinger påvirker studentenes programmering og hvilke arbeidsvaner de har.

Når jeg presenterer resultatene fra analysen vil jeg presentere funn fra både intervjuene, observasjonene av oppgaveløsning og spørreskjemaet i samme avsnitt. Jeg velger å gjøre dette siden jeg bruker sitater fra observasjonene og spørreskjemaet for å støtte opp sitatene fra intervjuene. Samtidig har jeg viet avsnitt 4.2 til å gi en kort redegjørelse for hvordan gruppene løste oppgavene. For å gi en oversikt over hvor resultatene kommer fra, blir sitater fra fokusgruppeintervjuene merket med *FGX* (X representerer nummeret på fokusgruppe 1-3). Sitater fra den observerte oppgaveløsningen merkes med *OGX* (X representerer nummeret på gruppe 1-3) og sitater fra spørreskjemaet blir merket med *SK*. Siden studentene er anonyme, blir de henvist til med hver sin faste bokstav. Studentene i gruppe 1 betegnes som A-B, gruppe 2 er C-G, mens gruppe 3 er H-K. Intervjueren har fått betegnelsen M i sitatene.

Først vil jeg i dette kapittelet presentere temaene og kodene jeg har laget i denne analysen. Deretter gir jeg en kort redegjørelse for noen funn fra studentenes arbeid med gruppeoppgavene. Resten av kapittelet blir delt inn i avsnitt etter temaene fra den tematiske analysen.

4.1 Temaer og koder i analysen

I den tematiske analysen så jeg at flere av kodene kunne plasseres under de ulike CT-komponentene i modellen til Shute *et al.* (2017). De CT-komponentene som kunne relateres til kodene ble derfor definert som *temaer.* I analysen har jeg ikke funnet resultater som dekker Iterasjon eller Generalisering og derfor er de ikke tatt med i presentasjonen her. De kodene som gav tilstrekkelig med funn, men ikke passet inn modellen har gitt grunnlaget for tre andre temaer; *Eksempelprogrammer, Biologi og programmering, og Arbeidsvaner.* Temaene og de ulike kodene som faller inn under dem er vist i tabell 3.

Tabell 3 – Forklaring av temaer og koder som er brukt i analysen

Temaer	**Koder**	**Beskrivelse av kode**
Problemnedbrytning	Skrive store programmer	Beskrivelse av arbeid med å skrive store programmer/få programmene til å virke
Abstraksjon	Utfordringer med mye informasjon	Vanskelig å forstå hva oppgaven spør om, forstå hva som skal settes inn i et program, store tekstoppgaver
	Strategi for å finne informasjon	Hvordan studentene finner ut hva oppgaven spør om
Algoritmer	Prøv- og feil	Bruke prøv og feil for å løse oppgaver
	Matematikk	Matematikk i undervisning, matematikk utfordringer, matematikk i programmering
	Programstrukturer[1]	Bruk eller plassering av programstrukturer
Feilsøking	Feilsøkingsrutiner	Feilsøking av programmet, reparasjon av feil
	Syntaks- og konseptuelle feil	Feil som skyldes skrivefeil eller manglende semikolon, o.l. Feil som skyldes feil plassering eller bruk av programstrukturer
	Logiske feil	Får feil svar, men programmet virker
Eksempelprogrammer	Eksempelprogrammer	Bruker eller henviser til tidligere oppgaver eller forelesninger
Biologi og programmering	Biologi og interesse for programmering	Biovitenskapelige problemstillingers påvirkning på studentenes interesse for programmering
	Biologi og vanskelighetsgrad i programmering	Biovitenskapelige oppgaver og deres påvirkning på programmering, på forståelse av oppgaven og på vurdering av svar
Arbeidsvaner	Gruppearbeid	Arbeid i grupper, samarbeid

[1] [1] Her definerer jeg programstrukturer som et overordnet begrep for alle funksjoner, lister, løkker, o.l. i Python.

Det er viktig å påpeke at resultatene ikke gir fullstendige beskrivelser av studentenes kognitive prosesser i de ulike komponentene til Shute *et al.* (2017). For eksempel handler *Algoritmer* om både design, parallellisme, effektivitet og automatikk. Gjennom den tematiske analysen her jeg derimot kun funnet resultater som kan knyttes til algoritmisk design, slik at de tre resterende kognitive prosessene ikke blir omtalt.

Studentene bruker begrepet «koder» i alle intervjuene når de snakker om å programmere. Jeg har valgt å omtale dette som «program» når jeg tolker og diskuterer resultatene, siden det er mer riktig å si at man «skriver ett program», enn «skriver en kode».

Fra spørreskjemaet ble det i hovedsak identifisert tre hovedutfordringer hos studentene. Av de 77 studentene som svarte, mente 29 at matematikk var mest utfordrende, mens 16 mente at riktig bruk av programstrukturer var mest utfordrende. 10 av studentene mente derimot at det å finne ut hva oppgaven ville de skulle frem til var mest utfordrende. Det ble også identifisert andre utfordringer, men her var det kun et fåtall svar som kunne relateres til utfordringene og de blir derfor ikke omtalt i denne studien.

4.2 Gruppenes arbeid med programmeringsoppgavene

I dette avsnittet vil jeg gi en kort analyse av oppgavene gruppene arbeidet med, og analysere forskjellen i arbeidet deres med de ulike oppgavene.

Alle studentene klarte å løse oppgave a uten store problemer. Gruppe 2 og 3 løste den fint på egenhånd, men gruppe 1 var litt usikre på genotypen til besteforeldre-generasjonen og måtte spørre om hjelp. Studentene i gruppe 1 hadde funnet riktig genotype til besteforeldrene, men var usikre fordi de fire besteforeldrene kunne ha to forskjellige genotyper. Løsningsforslagene til oppgavene ligger som vedlegg D.

OG1:
«A: Besteforeldrene kan jo være både, altså ene kan være dominant, andre kan være bærer, men begge kan og være heterozygote, så vi lurer litt på hvordan vi skal sette opp sannsynligheten for det.

B: For det eneste vi vet er jo at hverken foreldrene eller besteforeldrene viser noe tegn på sykdommen, men de kan, en av de må jo i hvert fall være bærer, om ikke begge.»

Her virker det som at studentene har et programmeringsrettet-fokus og prøver å forstå hvordan de kan lage en likning ut fra genotypene. Studentene blir da usikre fordi de trenger å vite sannsynligheten for arv av de ulike allelene og dermed ikke vet hvilken genotype som er riktig. At studentene i fokusgruppe 2 går inn i oppgaven med et programmeringsrettet fokus blir også tydelig i sitatet under.

OG2:
«E: De er bærere. Hvilken genotype har vi da?

C: Enten homozygot, eller, for, recessivt, eller heterozygot. (…)

E: Nå henger jeg ikke med likevel.

G: Nå vet jeg ikke hvordan jeg skal kode her.

E: Jeg vet ikke hvordan vi skal kode jeg heller.»

Som vi ser i sitatet strever studentene med å forstå hvordan de skal klare å programmere for genotypene, ettersom besteforeldrene potensielt kan ha to genotyper.

I oppgave b skulle gruppene finne svaret på sannsynligheten for å få tre syke barn. Det riktige svaret var at det var 1/729 dels sjanse for å få tre syke barn (se vedlegg D). Alle tre gruppene regnet seg derimot frem til at sannsynligheten for tre syke barn var 1/64.

OG1:
«B: Jo, men tenk nå, hvis begge foreldrene er bærere.

A: Ja.

B: Begge, fordi begge foreldrene må være bærere. Og for at de skal få tre syke barn, så er det jo en fjerdedels sjanse for hver gang. Og en fjerdedel ganger en fjerdedel, en sekstendel»

Studentene viser at de behersker den matematiske delen av oppgaven, da de viser at de forstår hvordan de skal regne med sannsynligheten for å arve begge de recessive genene fra foreldrene. De viser også forståelse av at de må ta med sannsynligheten for at foreldrene også

blir bærere. Det studentene glemmer å ta i betraktning er at hverken foreldre eller besteforeldre er syke i oppgaven og derfor skal de ikke ta med sannsynligheten for at foreldrene blir syke i regnestykket. Jeg tenker at det kan være to årsaker til at studentene fikk feil svar her. Den ene er at studentene ikke har tilstrekkelig biologikunnskaper til å forstå at de skal ekskludere sannsynligheten for syke foreldre. Den andre er at de misforstod oppgaven, noe vi ser i gruppe 2, som mente at oppgaven var forvirrende og at dette førte til at de fikk feil svar.

FG2:

«F: Altså, jeg tror at hadde de, vært liksom, vi ble litt misleada, eller på norsk, eh, feil, vi begynte jo i sånn feil retning fordi vi hadde sånn, antakelse, eller sånn fra før. (…)

F: Fra tidligere oppgaver og fra oppgaveteksten»

Med forklaringen til gruppe 2 virker det som at de baserer kunnskapen sin på tidligere oppgaver. Dette kan tyde på at studentene sitter med overfladiske biologikunnskaper og at de reproduserer svar fra tidligere oppgaver fremfor å forstå prinsippene bak arv og sannsynlighet i biologien.

I oppgave c skulle gruppene lage et program som genererte genotypen og fenotypen til fem barn der foreldrene var bærere. Detaljene rundt studentenes programmeringsarbeid blir drøftet mer inngående i de resterende avsnittene. Det jeg fokuserer på her, er hvordan studentene opplevde vanskelighetsgraden av oppgave c. Alle tre gruppene løste denne oppgaven uten store problemer. Gruppe 1 brukte litt mer tid enn de to andre gruppene, men det kan skyldes at de kun var to studenter på denne gruppen. Når gruppene blir spurt om hvilken oppgave de syntes var vanskeligst svarer gruppe 1 og 3 at oppgave c var mest utfordrende. Samtidig trekker både gruppe 2 og 3 inn at det gikk lettere enn forventet å løse oppgave c.

FG3:

«H: Jeg vil jo si at programmeringen, delen var jo vanskeligst, men det kanskje, gikk overraskende lett følte jeg. (…)»

FG2:

«F: Men, men programmeringen gikk jo og veldig greit da.

C: Ja, det gjorde det.

E: Ja, programmeringen gikk veldig greit faktisk.»

Her ser vi en sammenheng mellom hvordan studentene opplevde oppgavene de løste, og hva de føler de mestrer i BIOS1100. De fleste studentene i gruppe 2 og 3 føler at de mestrer biologien best. To av studentene i gruppe 2 forteller derimot at de opplever størst mestring i programmeringen.

FG2:

«M: Er det noen deler av dette emnet dere føler dere mestrer mer enn andre ting?

D: Biologien.

F: Kodingen.

C: Biologien.

E: Ja, biologien.»

Samtidig så er det noen indikasjoner på at oppgavene de løste var enklere enn forventet. Dermed kan det være at oppgavene de løste ikke er helt representative for det de arbeider med ellers i BIOS1100, til tross for at oppgaven er hentet fra BIOS1100s eget kursmateriale. Samtidig påpeker gruppene at programmeringsoppgaven lignet veldig mye på det de vanligvis arbeidet med i emnet, men at hele oppgavesettet, da spesielt a og b, var mer biologisk rettet enn de var vant med.

FG2:

E: Ja, jeg tror at a og b var BIOS1110 og c er jo BIOS1100, så det var vel kanskje et eksempel på hvordan en kan kombinere b og c litt.

FG3:

«M: Hvordan var den i forhold til det dere pleier å jobbe med i kurset da?

H: Jeg føler den var mye mer rettet mot biologi, egentlig.

J: Mhm

I: Mm, mye mer sånn det vi holder på med i år.

J: Ja. Jeg føler det vi har jobbet med tidligere i programmeringen angående biologi er på en måte litt mer, ja.

I: Litt annerledes»

Sistnevnte kommentar fra fokusgruppe 2 kan tyde på at BIOS1100 er mest programmeringsorientert og at oppgaven de løste under min datainnsamling var mer biologisk orientert enn hva de var vant med. Dette kan også forklare hvorfor studentene startet oppgave a med et programmeringsrettet-fokus.

4.3 Resultater fra fokusgruppeintervju

4.3.1 Problemnedbrytning – Dele oppgaven i mindre deler

Den første kategorien til Shute *et al.* (2017) er problemnedbrytning. I dette avsnittet ser jeg på hvordan studentene opplever oppgaver som er store og hvordan de arbeider med dem.

En utfordring to av fokusgruppene forteller om er å lage store programmer. Studentene trekker frem at det er to ting som er utfordrende med dette. Det første er at når programmet blir stort, er det vanskelig å holde styr på alt som skal med i programmet.

FG1:
«A: Ja, så det er liksom det å sette sammen alt det. Ofte blir det jo en celle for eksempel, og få til alt til å, heh, henge sammen. Og ville jobbe sammen.»

FG2:
«E: Å holde styr på alt.»

Det andre som gjør det utfordrende å lage store programmer er å få programmene til å kjøre slik de ønsker.

FG1:
«A: Ja, så det var liksom det, å klare å lage en lang kode som faktisk får til det en ønsker. Og bruk av de ulike while og if og sånn.»

I dette avsnittet velger jeg å ikke gå i dybden av hvorfor studentene har utfordringer med store programmer. Dette er et tema som er mer relevant for den algoritmiske delen av CT og vil derfor presenteres i avsnitt 4.3.3. Det som derimot er tydelig her og som er relevant for

studentenes problemnedbrytning, er at oppgaver med mye informasjon og oppgaver som krever at det lages store programmer, er utfordrende for studentene. At studentene opplever dette som utfordrende kan indikere at de har dårlige strategier for å bryte ned problemene i oppgavene. En av respondentene i spørreskjemaet forteller for eksempel at det er vanskelig å vite hvordan man starter å programmere i store oppgaver.

SK:
«Det jeg synes er vanskelig er hvordan man skal gå frem i starten av store oppgaver»

Både under intervjuet og ved observasjon av oppgaveløsningen, viser studentene at de har en rekke strategier for å løse programmeringsoppgaver. Hva disse strategiene er, kommer jeg tilbake til senere i kapittelet. Det som derimot er tydelig, er at ingen av strategiene kan relateres til problemnedbrytning. Når studentene starter å løse oppgave c i oppgavesettet, starter de rett på programmeringen, uten å dele opp oppgaven.

OG2:
«F: Men denne her er ikke så vanskelig å programmere, den c en skal gå greit. For da, hvis vi tar, det er femti prosent sjanse for at, eh, altså, at de får stor r eller liten r fra hver forelder, så da kan vi bare bruke sånn choice, kan vi ikke?

C: Mmm, ja.

F: Vi har en stor r og en liten r, også bruker vi bare choice til å velge.

C: Da må vi lage en liste da. Kan vi ikke det.»

At studentene ikke bryter opp programmeringen i oppgave c må ikke bety at de mangler strategier for å bryte opp problemet. Som vist i løsningsforslaget til oppgave c (Vedlegg D) trenger ikke programmet å bli spesielt stor, og derfor er det ikke nødvendig å bryte opp denne oppgaven. Likevel kan det se ut som at det er en sammenheng mellom studentenes opplevde utfordring med å lage store programmer og deres manglede bevissthet rundt det å dele opp problemet i mindre deler.

4.3.2 Abstraksjon – Å finne ut hva oppgaven spør om

Abstraksjon er kategorien til Shute *et al.* (2017) som handler om å trekke ut relevant informasjon, avdekke trender og mønster i oppgaven og modellere et system som kan programmeres. Det første jeg presenterer her går inn underkategorien til abstraksjon, som er datainnsamling og analyse.

Når fokusgruppene snakker om hvordan de løser programmeringsoppgaver påpeker alle tre gruppene at en del av arbeidet involverer å ta viktig informasjon ut av oppgaveteksten. Alle gruppene er også klare på at noe av det viktigste de må gjøre når de programmerer er å finne ut hva oppgaven vil fram til.

FG2:
«F: Det viktigste er jo å vite hva du skal frem til.
G: Hva du har av informasjon og hva du skal bruke det til.
C: Mm. Ta ut de viktige bitene.»

To av gruppene påpeker at noe av utfordringen med programmering er å finne ut hva oppgavene spør om.

FG3:
«4: Ja. Jeg føler sånn, på programmeringsoppgavene så er det ofte vanskelig å finne ut hva de spør om»

I spørreskjemaet svarte 10 av 77 respondenter at det var mest utfordrende å forstå hva de skulle programmere i en oppgave. Denne utfordringen utgjorde derfor en av de tre største utfordringene med å programmere.

SK:
«Vanskelig å forstå hva man skal gjøre»

«Å forstå hva oppgaven ber meg om å programmere.»

De siste sitatene viser at det kan være en utfordring å forstå hva programmeringsoppgavene spør om. Dette kan ha en sammenheng med at studentene påpeker hvorfor det er viktig å finne ut hva oppgaven er ute etter. Studentene i fokusgruppe 1 snakker også om at hvis de er usikre på hva de putter inn i et program, så blir de usikre på om resultatet fra programmet blir riktig.

FG1:

«B: Da vet man jo heller ikke om det man får ut er riktig. Hvis man er usikker på det man setter inn.

A: Nei. Man vet ikke hva en skal få ut heller liksom.»

Jeg tolker at denne utfordringen med å trekke ut relevant informasjon i programmeringsoppgaver har en sammenheng med at selve programmeringen blir et mellomledd for å få et svar i oppgaven. Studentene må ikke bare lage et program, men de må passe på at den informasjonen som settes inn i programmet er riktig for at svaret de får ut av programmet skal være korrekt.

Man kan også se at biologiske problemstillinger spiller en rolle i studentenes arbeid når vi ser det i lys av abstraksjonsfasen til CT. Generelt har studentene litt ulike meninger om dette. To av fokusgruppene forteller at biologiske problemstillinger kan være nyttige, da det kan gi studentene en pekepinn på hva svaret skal bli i oppgaven. Studentene forteller at når de har en biologisk problemstilling så kan de bruke det for å se om svaret de får av programmet er korrekt eller ikke.

FG2:

«E; Ja, det kan godt hende, at man blir litt sånn, åja, det svaret der burde være rett liksom, eller sånn, det svaret her det ser rett ut. Fordi du vet hva det egentlig skal være, sånn biologisk sett. (…).

M: Dere da?

F: Nei, litt det samme. Jeg vil si det er litt enklere når jeg ikke har det (biologi i oppgaven). I de fleste tilfeller. Men så er det noen ganger der det hjelper. Vi hadde jo, vi skulle plotte, en, noe med temperatur, da visste vi at den skulle gå opp, og så skulle den gå ned. Og så fikk jeg først egentlig bare at den gikk rett opp, og da så jeg at det var feil (…).»

FG1:

«A: Men samtidig så har vi jo litt pekepinn på det, men oppgaven var jo egentlig, den var krevende men grei.»

«F» forteller her at han syntes det er enklere å programmere når det ikke er biologi i oppgaven, samtidig forteller han at det noen ganger kan hjelpe å få en pekepinn på hva svaret skal bli. På en side kan det tenkes at biologiske problemstillinger krever at studentene må tenke mer for å finne ut hva de skal sette inn i programmene sine. Samtidig kan biologiske

problemstillinger kan gjøre det lettere for studentene å modellere et system som de kan lage et program av. Jeg tolker at det er tilfelle, fordi studentene påpeker at den biologiske informasjonen gir et utgangspunkt for hva de kan forvente å få i svar når de kjører programmene sine.

Resultater fra fokusgruppe 2 og 3 viser at studentene opplever oppgaver med mye tekst som krevende. Flere av studentene forklarer at dette skyldes at det er så mye informasjon å forholde seg til samtidig.

FG2:
«D: Deler var greit, men da vi kom til steder der alle var usikre så var det litt vanskelig å tenke egentlig, for det var så mye informasjon på en gang.»

FG3:
«H: Nesten litt sånn overkomplisert. Det er så mye informasjon»

Når det gjelder programmeringsoppgaven studentene arbeidet med, var det uenighet innad i fokusgruppe 2 om hvorvidt teksten var relevant for oppgaven eller ikke. Studentene var enige om at det var mange detaljer å forholde seg til i oppgaven, men noen av studentene mente at teksten var irrelevant for det de skulle programmere, mens andre mente den var relevant.

FG2:
«C: At i starten her var det litt unødvendig egentlig»

«E: Altså, det er unødvendig hvis en ikke vet noe om det, men sånn det er jo ikke unødvendig hvis man har glemt det,

G: Det var overflødig for oppgaven.»

Når fokusgruppene ble spurt om hvordan de jobbet for å finne ut hvilken informasjon som var viktig i en programmeringsoppgave, viste de til forskjellige strategier. To av fokusgruppene fortalte at de leste oppgaveteksten flere ganger og prøvde å peke på ting som var viktige, eller som fanget oppmerksomheten deres. En av fokusgruppene trakk også frem at når de arbeidet med krysningsskjemaer i oppgave a, prøvde de å sirkle rundt hva som skulle gjøres i hver oppgave, og at de lagde lister over all informasjonen de hadde.

FG3:
«I: Man leser gjennom spørsmålet, eller, først leser oppgaven, også spørsmålene, også lese på nytt en gang.

K: Ja.

I: Ja. Lese en gang til, og så bare prøve å peke ut det viktige.

J: Prøve å liksom, filtrere det som oppgaven spør om.

I: Ja.

J: Mhm. Så det blir bare å lese mange ganger.»

FG2:
«C: Lese teksten flere ganger. Se om det er noe som fanger oppmerksomheten.»

FG1:

«B: Ja, jeg ville skrevet ned en liste med all informasjon først og så begynt å se, og så laget noen sånne punnet skjemaer(kryssningsskjema) og fått tak i det jeg visste fra teksten.

A: Ja. Sirklet litt rundt hva som skulle gjøres i hver oppgave, eh, ja.»

Sitatene viser at studentene har litt forskjellige tilnærminger til å finne ut hva oppgaven er ute etter. Det som går igjen i to av gruppene er at de legger vekt på å lese oppgaven flere ganger. Dette kan tyde på at studentene ikke har så mange bevisste strategier for å trekke ut den relevante informasjonen fra teksten, da det kun er studentene i gruppe 1 som viser til en konkret strategi for dette, nemlig å skrive ned en liste.

4.3.3 Algoritmer – Fra tekst til program

Komponenten som omhandler algoritmer i Shute *et al.* (2017) sin modell handler om å lage en stegvis instruksjon, ofte en likning, for så å implementere den i et program. I denne kategorien ser jeg på funn knyttet til studentenes arbeid med likninger i programmeringen.

Når det kommer til å skrive et program forteller gruppene at det ofte er utfordrende å sette den sammen på en korrekt måte. De tre gruppene trekker frem flere grunner til at de opplever nettopp dette som utfordrende.

FG3:

«H: Sånn, ser på tidligere ting man har gjort. Fordi det er jo så mye detaljer og sånn. Så har man en feil så blir jo alt feil.»

FG1:
«A: Det var ganske mye som skulle inn i en celle, og på en måte, ja få snørt han sammen, og litt sånn. Men samtidig når en først får det til, så gikk det bra. Så det var på en måte, ja.

B: Ja, for der var det så mange ulike puslespillbiter som skulle passe sammen. Så det ble en kjempelang kode.»

Gruppenes utsagn tyder på at detaljnivået i programmeringen påvirker hvor utfordrende det er å programmere. Studentene i gruppe 2 påpeker at det er veldig mange detaljer som skal være med for at programmet skal virke. I forhold til hvilke detaljer studentene snakker om, forteller to av gruppene at de synes det kan være vanskelig å vite hva alt gjør, og hvorfor ulike ting må skrives som de skal. Nøyaktig hva de mener med «alt» må tolkes. Begge gruppene trekker både inn løkker, så vel som funksjoner, så jeg velger å anta at det er snakk om de ulike programstrukturene i Python.

FG2:
«F: (…) Altså få alt på riktig sted sånn at, noe skal være inni og noe skal være utenfor en løkke.
E: Ja, ikke sant. For det har jo også veldig mye å si.»

«C: Ja. Det er vel det å skjønne hva alt gjør og hvordan, hvorfor må man skrive sånne ting som man må.»

FG1:

«A: Ja. Det er jo på en måte, hva skal jeg si, sette sammen, alle disse, hva heter det, disse while og funksjonene. Og disse ulike greiene. Og sette dem sammen. Skal du ha while først, og så plutselig går det ann å bruke 2 funksjoner. (…)»

En av hovedutfordringene som ble identifisert i spørreskjemaet, var knyttet til å lage programmer ifølge 16 av 77 respondenter. Av de 16 svarene, var 10 knyttet til programstrukturer, spesielt *while*-løkker. Dette resultatet, sammen med fokusgruppeintervjuene tyder på at den konseptuelle kunnskapen i programmeringen er en utfordring for studentene (Bayman & Mayer, 1988). Ser man på sitatene fra gruppe 1, viser de

et nokså upresist språk om programstrukturene i Python. Dette bidrar til å styrke denne tolkningen, da det upresise språket kan være en konsekvens av manglende konseptuell kunnskap hos studenten.

I fokusgruppe 3 forteller flere av studentene at det er vanskelig å vite hvordan de skal begynne å programmere.

FG3:
«I: Jeg føler det alltid er vanskelig å vite hvordan en skal begynne på en måte.

H: Mhm

I: Hva man skal begynne med og sånn»

«M: Ja, okay, så det er det at det er vanskelig å begynne med, er det noe annet som gjorde det vanskelig å jobbe med programmeringen?

J: Det er jo veldig mange forskjellige koder og sånne ting. Det er vanskelig å vite akkurat hvilken man skal bruke.»

Som jeg har vist tidligere i dette avsnittet er detaljnivået i programmeringen en utfordring for studentene. Det virker som det er krevende å ha oversikt over alle de ulike detaljstrukturene, men også å klare å sette de ulike strukturene sammen til et program. Studentens utfordring med å starte å programmere synes å være en konsekvens av manglende forståelse av de ulike programstrukturene i Python. Når de ikke har oversikt over hva og hvordan de setter sammen strukturene i et program, blir det dermed vanskelig å forstå hvordan de skal starte.

En av studentene i gruppe 2 forteller også at en utfordring med å programmere, er at løsningene de lager til oppgavene kan være forskjellige.

FG2:
«E: Det er det jeg føler at det er veldig irriterende med dette faget. Jeg hater å ha flere løsninger.

F: Gjør du det?

E: Ja, kan vi finne et fasitsvar, er det ikke derfor vi driver med vitenskap? Vi skal prøve å finne et best mulig svar»

«E: Ja. Jeg tror kanskje det også huske hva alt gjør, det kan bli litt forvirrende noen ganger. For plutselig så har du skrevet en kode også er det, er du vant til å skrive inn en linje liksom, også skal det egentlig ikke være med på neste program da…»

Studenten gir uttrykk for frustrasjon over at det er så mange måter å løse en programmeringsoppgave på. Denne frustrasjonen ser ut til å ha to årsaker. For det første så trekker studenten frem at mengden ulike løsningsmuligheter blir forvirrende. Som det siste sitatet viser, lærer studentene seg å lage programmer på en bestemt måte, men så må de skrive det annerledes i et nytt program. Dette gir en indikasjon på at når studentene programmerer, har de ikke de grunnleggende kunnskapene som gjør dem i stand til å lage et program på egenhånd. I stedet virker det som at de lærer seg hvordan bestemte programmer skal se ut og forventer at de kan bruke det samme programmet i andre oppgaver. Den andre årsaken til studentens frustrasjon ser ut til å være at studentene forventer at det skal være ett riktig svar. Studenten har en forventning om hvordan vitenskapelige oppgaver skal være, og blir frustrert når dette ikke samsvarer med denne forventningen. Her kan det tenkes at vedkommende sammenlikner programmeringsemnet med andre biovitenskapelige temaer, hvor man ofte lærer studentene én måte å løse et problem på. Eksempelvis, lærer studentene kun å lage krysningsskjemaer når de gjør oppgaver om arv og sannsynlighet i biologien.

En del av det algoritmiske arbeidet er å formulere en likning, for så å skrive den om til et program. Fra spørreskjemaet svarte 28 av 77 respondenter at matematikken var mest utfordrende og var den utfordringen som ble mest omtalt. Alle tre fokusgruppene forteller også at de opplever matematikken som utfordrende når de programmerer.

FG2:
«D: Jeg syntes matten er veldig vanskelig i tillegg.»

FG3:
«K: Jeg føler at matten er det vanskeligste.»

Matematikken trekkes frem som en utfordring av flere studenter her. Det er ikke bare det programmeringstekniske som er utfordrende, men også det å anvende matematikk når de programmerer. Likninger er et tema som blir tatt opp ved flere anledninger i intervjuet. Studentene i gruppe 3 forteller at det er vanskelig å skrive egne likninger. De trekker også frem at det er vanskelig å få programmet til å oppfatte likningen riktig.

FG3:
«H: Ja, det syntes jeg og. Det er sånn, det er veldig greit når de gir oss likningen, men når de sier sånn, finn likningen selv, så blir det sånn, å herregud, hvor skal jeg starte.

J: Ja.

H: Jeg føler programmet liksom oppfatter likninger på en helt annen måte enn når vi gjør det på ark på en måte.

J: Ja.

K: Ja.

H: Vi må jo tilpasse likningen til programmet. Og det vet jeg ikke helt hvordan man gjør. Men men»

Det er tydelig at de matematiske kunnskapene har en sentral rolle i BIOS1100. Når studentene blir spurt om hva slags forkunnskaper de har bruk for i emnet, viser to av gruppene til ulike mattekunnskaper som likninger, funksjoner, rekker og logaritmer.

FG3:
«M: Sånn som hva da? Har dere noen eksempler?

I: Likninger, funksjoner.

J: Logaritmer.

K: Litt rekker og sånn.»

En del av forelesningene i BIOS1100 er rettet mot den matematiske delen av programmeringen. Flere av studentene uttrykker at de opplever disse forelesningene som utfordrende.

FG3:
«I: Hvertfall når vi har hatt forelesninger om matte, for min del så skjønte jeg ingenting.»

FG1:
«B: Ja. Og de første forelesningene vi hadde i matte da var det jo nesten så han gikk utfra at alle hadde R2 matematikk.

A: Ja

B: Og det var et fåtall. Jeg skjønte ikke hva han snakket om i det hele tatt. Og nei, det var litt skremmende»

Det student «B» forteller, indikerer at matematikkforelesningene i BIOS1100 ligger på et nivå som forutsetter at studentene har hatt R2 matematikk. Som vedkommende sier, er det ikke alle studentene som har det. BIOS1100 har ikke hatt R2 opptakskrav, så mange av studentene har ikke vært igjennom pensum i R2 matematikken. Samtidig forteller en av studentene at til tross for at vedkommende har hatt R2, så har ikke det hjulpet med matten i BIOS1100.

FG3:
«H: Også, jeg har jo, for meg som har hatt R2 så er det litt gøy å vite at det kan være nytte for det, men jeg føler ikke at min R2 undervisning har hatt noen nytte for seg.»

Denne analysen viser at mange av studentene ikke har tilstrekkelige forkunnskaper til å møte den matematikken de arbeider med i BIOS1100, og derfor oppleves matematikken som spesielt utfordrende i programmeringen. Når studentene sliter med å forstå matematiske konsepter som likninger, blir det også vanskeligere for dem lage likninger når de programmerer.

Fra intervjuene kommer det tydelig frem at studentene prøver seg mye frem når de programmerer.

FG1:
«M: Kan dere ikke ta meg gjennom prosessen. Hvordan gikk dere frem for å løse oppgaven?

B: Prøving og feiling.»

FG2:
«F: Nei, jeg tenker mer på sånn røff sketsj av greia, og så etter det blir det en sånn prøving og feiling. Hva passer best og sånn.»

FG3:
«H: Ja. Man spør litt, egentlig, man prøver, det er sånn, man prøver litt selv, ser litt på sånn tidligere oppgaver man har gjort på sånn livekoding og sånt..»

«H: Ja. Det er jo litt sånn som vi alltid gjør programmering føler jeg. Prøve og feile type.»

Student «H» påpeker at det er ofte typisk for programmeringen at de prøver seg frem, og det virker som at det er en generell aksept blant gruppene på at det er sånn de jobber med programmering. Sitatene gir uttrykk for at studentene ser på tidligere oppgaver og prøver å tilpasse dem til nye problemstillinger. Denne «prøv- og feil» mentaliteten til studentene tyder på at de ikke har andre, mer effektive strategier for å løse programmeringsoppgaver.

4.3.4 Feilsøking – Hvordan løser studentene feilmeldinger

Feilsøking fra modellen Shute *et al.* (2017) handler om å identifisere og reparere feil når programmet ikke virker som det skal. I det forrige avsnitt viste jeg hvordan studentene bruker en prøv- og feil-strategi for å skrive programmer. I tillegg til å være en mye brukt strategi for å lage programmer, forteller også to av fokusgruppene at de bruker denne strategien når de får opp feilmeldinger i programmet. Prøving og feiling blir en naturlig del av feilsøkingen, der studentene kjører programmet flere ganger for å undersøke om den fungerer eller ikke.

FG1:
B: For det er jo, for min del i hvert fall, koding er en del prøving og feiling.

A: Ja, meg og. Veldig mye. Jeg kjører og kjører og kjører. Og retter opp og retter opp.»

FG3:
«K: Men det har vel bare med at vi kjører koden for å sjekke om den funker før den er ferdig da.

J: Ja.

I: Ja.

H: Så vi tar egentlig veldig nytte av det å kjøre koden.»

I det siste sitatet til fokusgruppe 3 påpeker den ene studenten at de har stor nytte av å kunne kjøre programmene. Dermed kan det virke som at prøv- og feil-strategien er et verktøy studentene mer eller mindre bevisst bruker når de programmerer. Som nevnt i informasjonen om BIOS1100, så har ikke studentene muligheten til å kjøre programmet de lager på eksamen. Studentene i alle tre fokusgruppene forteller at de bekymrer seg for eksamen, da de ikke kan kjøre programmene. Det er derimot ulike meninger om akkurat dette blant studentene. I fokusgruppe 2 mener studentene at prøving og feiling er en viktig del av programmering. De uttrykker seg negativt til at eksamen ikke gir dem denne muligheten. I fokusgruppe 1 forteller

derimot studentene at de glad for å ikke kunne kjøre programmet, da det kunne ført til at det låste seg helt for dem om de fikk feilmeldinger på eksamen.

FG3:
«H: Biologi er litt mer sånn, trygg grunn, kan man si. Mens programmering er mer sånn, prøv og feil.

I: Ja. Når vi får sjansen til å prøve å feile da. På eksamen så kan vi jo liksom ikke se om vi har gjort feil eller ikke»

FG2:
«G: Men det er jo litt teit at eksamen er lagt opp sånn at vi ikke skal kunne prøve å feile. En veldig viktig del av det er jo å prøve seg frem.

D: Det er jo ikke sånn at i det virkelige liv når vi sitter der og skal programmere at vi ikke får kjørt kodene.»

FG1:
«B: Ja, vi brukte jo faktisk det også nå. Også i tillegg så er det, det da får du kjørt programmet ditt også får du se hvor er det jeg har gjort feil, hvor er det jeg kan gå inn og rette opp. Selv om jeg er jo glad for at vi ikke kan kjøre koden på eksamen, fordi da tror jeg at jeg hadde gått helt i baklås, hvis jeg ikke hadde fått riktig.»

Fokusgruppene forteller også at de bruker andre metoder når de skal løse feilmeldinger. Studentene i fokusgruppe 1 viser litt ulik tilnærming til hvordan de løser feilmeldinger. Begge studentene forteller at de gjerne jobber seg bakover i programmet, og forsøker å finne ut hvor feilen ligger. En av studentene viser også til at hun ofte søker på nettet for å finne ut hva feilmeldingene betyr og hvordan de kan løses.

FG1:
«B: Jeg får veldig mange spennende sånne feilkoder, feilmeldinger hjemme, så jeg må stadig vekk søke de opp, for jeg får stadig vekk nye ting som jeg aldri har sett før.

A: Så du bare Googler de?

B: Ja, jeg Googler de. Innimellom så skjønner jeg tilbakemeldingene på nettet, andre ganger ikke. For det er jo skrevet av programmerere.

A: Jeg pleier ikke å Google. Jeg pleier liksom bare å gå igjennom prosessen, og liksom ofte prøver jeg å tolke hva jeg får printet ut og liksom, prøver å gå litt baklengs, liksom, hvor var siste feil oppstått.

B: Ja. For jeg ser etter det og, men når jeg ikke finner noe eller hvis jeg ser meg blind på det jeg har sittet og gjort, da søker jeg på feilmeldingen.»

I fokusgruppe 2 forteller studentene at de bruker feilmeldingene til å finne ut hvor feilen ligger, for deretter å prøve å reparere feilen.

FG2:
«D: Sjekker hvor feilen ligger hen, og prøver å finne ut hvordan man fikser det. Som regel for meg så er det semikolon som er glemt (latter), eller en skrivefeil»

Studenten beskriver litt ulike typer feil de gjør i intervjuet. Student «D» forteller at det som regel er snakk om et glemt semikolon eller en skrivefeil. Her skal det påpekes at semikolon ikke brukes i Python, men at kolon er mye brukt. Det er derfor mulig at studentene egentlig mener ett kolon. Flere av studentene i de andre gruppene forteller også at det er normalt at det blir skrivefeil når de lager programmene sine.

FG1:
«A: Ja. Som regel er det jo bare en skrivefeil. Ofte da, selvfølgelig ikke med et helt program.»

FG3:
«H: Eh, det var bare sånn slurvefeil.

K: Ja, altså i steder for å skrive print, så skrev vi prin. Fikk ikke med den t'en (…)

H: Så det var ikke noe spesielt. Det var ikke noe feil i koden sånn, selve måten vi gjorde det på. Det var bare skrivefeil. På en måte.»

Her er det litt usikkert hva studentene legger i ordet «skrivefeil». Studentene kan både mene at de skriver et ord feil, men også at de glemmer ting i programmet, som et kolon. Begge deler går derimot inn under det man kaller syntaktisk kunnskap (Bayman & Mayer, 1988) så her behandler jeg skrivefeil og syntaksfeil som det samme. Basert på studentenes utsagn, virker det som syntaksfeil forkommer relativt ofte hos studentene. Ser vi på sitatet til student «D», så virker det derimot ikke nevneverdig utfordrende for studentene, og det fremstår som at de er komfortable med å rette syntaksfeil.

Gruppe 3 presenterer også en type feil de fikk når de programmerte. Denne feilen var knyttet til plassering og bruk av programstrukturer som *choice* og *else*.

FG3:
«J: Ja. Det var sånn, litt sånn omrokkering av choice og litt sånn.

H: Ja.

K: Også glemte vi å legge til else og sånn».

Denne typen feil som beskrives her er typiske konseptuelle feil, som er feil som skyldes at de ulike programstrukturene ikke er satt riktig sammen i programmet (Bayman & Mayer, 1988). Sitatene forteller ikke så mye om hvordan studentene opplevde denne feilen, men måten de forteller det på, indikerer at de ikke opplevde det som spesielt krevende å rette opp denne typen feil heller.

Studentene i fokusgruppe 2 nevner også at de av og til får andre typer feil. Dette er feil der programmet virker som det skal, men svaret de får er feil. Her påpeker studenten at en slik feil gjerne ligger i logikken bak programmet. Dette er eksempel på logiske feil, som oppstår når programmet virker, men det programmet skriver ut et feil resultat (Hristova *et al.*, 2003). Den samme studenten påpeker at en slik feil ofte er vanskeligere å forholde seg til enn en skrivefeil, da det krever at de gjerne må begynne på nytt i programmeringen.

FG2:
«F: Altså, det spørs jo litt hvordan feilen er da, om det er feil i kodingen eller i logikken.

D: Ja.

E: For det er ofte at man får feil, også bare får en et helt feil svar. At det er ikke noe galt med selve koden, men at du får ikke riktig. Da må du jo, begynne på nytt, på en måte. Du må liksom gå tilbake til begynnelsen og se litt hvor, hva var det som ikke stemmer her. Litt mer irriterende enn når det bare er en, enkel sånn at den bare viser at der er feilen.»

Analysen i dette avsnittet viser at studentene møter forskjellige typer feil når de programmerer. Her virker det som at det er syntaksfeil og skrivefeil er de mest vanlige, men at også konseptuelle feil forekommer. Studentene synes dog ikke å stresse nevneverdig med disse feilene, og ser ut til å vise god kompetanse i å rette dem opp. Som det siste sitatet til student «E» viser, opplever de også logiske feil. Denne typen feil ser ut til å være mer

krevende for studentene, da de ikke får opp feilmeldinger, men må jobbe seg gjennom programmet for å finne ut hva som gjør at svaret blir feil.

Resten av resultatene blir nå presentert utenfor CT modellen til Shute *et al.* (2017). Det betyr at det ikke blir presentert funn om Iterasjon eller Generalisering.

4.3.5 Bruk av eksempelprogrammer

Ett av temaene som jeg identifiserte gjennom analysen var det som omhandlet eksempelprogrammer. Studentene forteller at de ofte ser på tidligere oppgaver når de programmerer.

FG1:
«B: Som du sa i sted, så tidligere i hvert fall så har jeg pleid å bruke hva vi har gjort før og gå inn og se på det. Og det var jo det vi gjorde nå og. Vi gikk inn og så på det vi gjorde i timen blant annet.

FG2:
«E: Ja, det blir jo det. Også for å finne sånn kodeinformasjon så må en egentlig bare tenke tilbake på hva man har gjort og om man har hatt lignende oppgaver. Og hvordan man kan anvende det på den nye oppgaven her.»

FG3:
«K: Også går man jo ofte tilbake til gamle oppgaver hvor man har fått det til da
I: Mhm.
J: Ja, ser på mye gamle oppgaver
H: Sånn, ser på tidligere ting man har gjort. Fordi det er jo så mye detaljer og sånn. Så har man en feil så blir jo alt feil.»

Som det kommer frem i sitatene gitt ovenfor her, bruker studentene tidligere oppgaver for å huske hvordan de ulike programstrukturene virker, og for å klare å lage nye programmer. På en side kan dette være en strategi studenten bruker for å spare tid når de programmerer. Men, som det neste sitatet viser, er det større sannsynlighet for at det skyldes at studentene er usikre på hvordan de skal lage et program fra bunnen av.

FG3:
«H: Vi måtte jo se tilbake på tidligere oppgaver, sånn der vi vet vi på en måte har brukt dem også, for vi husker jo, sånn tilsvarende funksjoner på en måte (…)

H: Så, jeg vet ikke hvordan vi klarte det jeg»

Samtidig så er det mye som indikerer at studentene trenger å se tilbake på gamle oppgaver, fordi de er usikre på hvordan de skal skrive programmene sine. Dette blir spesielt tydelig i fokusgruppe 3, der student «H» påpeker at det er så mange detaljer å forholde seg til for å skrive et program. Som vi ser i det siste sitatet fra «H», forstår egentlig ikke studentene helt de klarte å lage programmet i oppgave c. Denne tolkningen bekreftes også av fokusgruppe 1, når jeg spør dem om hva de gjør hvis de må programmere uten hjelpemidler.

FG1;
M: Ja, hvis dere hadde sittet uten hjelpemidler, og dere skulle sett på selve kodedelen av det.

A: Ja. Da hadde det nok ikke gått så bra. For veldig mye, når jeg både løser dette, obliger og excercises og sånt så ser jeg veldig mye på hva jeg har gjort tidligere, hva vi har gjort i livekodigen i timen.

Student «A» forteller er at de henvender seg mye til tidligere oppgaver når de arbeider med programmering. Dermed virker det som at studentene opplever det som utfordrende å måtte løse oppgaver og lage programmer uten å kunne henvende seg til ting de har arbeidet med tidligere.

4.3.6 Rollen til det biologifaglige i programmering

Det er også litt delte meninger om hvordan biologiske problemstillinger i programmeringen påvirker interessen for programmering. Både studentene i fokusgruppe 1 og 3 forteller at de blir mer motiverte og interesserte i programmering når de ser sammenheng mellom det og biologien de arbeider for.

FG1:
«A: Ja. Men så er det og litt motiverende når en på en måte ser at programmeringen og biologien henger så sammen da.

FG3:
«K: Jeg tenker jo sånn, når vi må ha programmering, så er det fint å ha om biologi og ikke et rent programmeringsfag.

J: Ja

H: Ja, det er sant. Det hadde vært enda mindre interesse i programmering da.»

Funnene viser en klar sammenheng mellom at biologi og programmering er viktig for de fleste studentene. Det virker på studentene som at de ikke er spesielt interessert i å ha programmering i utgangspunktet, men at når de først må ha det, hjelper det at biologien har en plass i programmeringen. De neste sitatene viser også at studenten blir ekstra motivert for å lære seg programmering når de vet at de får bruk for det i biologen. Her ser det ut som at nytteverdien av programmering bidrar til motivasjonen for å lære det.

FG2:

«E: Samtidig så er jeg glad at vi har programmering her. Fordi det vil være veldig relevant i fremtiden.

D: Det er jo litt kult.»

FG1:
A: At en på en måte trenger programmeringen veldig til biologien. Så det er en motivasjon til å lære seg dette ordentlig da. At en helt sikkert har bruk for det.»

I fokusgruppe 2 er det derimot litt delte meninger. To av studentene forteller at de ikke blir spesielt interesserte eller påvirket av å ha biologiske problemstillinger, mens de resterende studentene er enige med fokusgruppe 1 og 3.

FG2:
«E: Eh, oppgaven gjorde vel egentlig ingenting, for meg. Jeg syntes det er helt greit, men det er ikke, jeg kommer ikke til å undersøke noe mer nå. Jeg må jo lære det, fordi vi har det i BIOS1110, men, ikke så mye mer utenom det.

F: Jeg er egentlig enig. Jeg syntes det mest spennende med faget er programmeringen. Og lære om hvordan det henger sammen. Mens det biologiske kommer litt etter, og jeg det har jeg liksom nok av i andre fag. Opplever jeg.»

De studentene som ikke interesserer seg for programmeringen, viser en noe forhøyet interesse for det når det er en sammenheng mellom programmeringen og biologien. For de to studentene som derimot var interessert i programmering virket det som at de var mindre interessert i å ha biologi som en del av programmeringen.

Når det kommer til hva studentene mener om å nytten av å ha biologiske problemstillinger som en del av programmeringen, er de stort sett enige i fokusgruppene om at biologien ikke har så mye å si for selve programmeringen. Programmene de skriver kan ifølge studentene brukes i hvilken som helst sammenheng og det har ikke noe å si om de programmerer med hensyn på biologi eller klinkekuler.

FG2:
«E: Ja, jeg føler at biologien ikke har så veldig mye å si når du koder da. Sånn egentlig. Sånn så lenge du vet sånn statistisk sett, hvilke muligheter du har, så spiller det ingen rolle om det handler om biologi eller om det handler om klinkekuler på en måte.»

«E:(...) Fordi du vet hva det egentlig skal være, sånn biologisk sett. Men sånn selve kodehandlingen og kode, det er egentlig det samme.»

FG3:
«1: Men, det kunne jo vært hvilken som helt oppgave, og bare hatt sammenheng med noe helt annet. Noe samfunnsfaglig eller, vi hadde jo liksom brukt de samme funksjonene uansett.»

Det virker her som at biologikunnskapene ikke har mye å si for studentenes arbeid med å skrive selve programmet. Slik jeg tolker utsagnene til student «E», er det sånn at så lenge du har forutsetningene for å svare på oppgaven, har det ikke noe å si hva du programmerer. Dermed virker det ikke som vanskelighetsgraden av å skrive et program endres hvis du gjør det med en biovitenskapelige problemstilling.

4.3.7 Studentenes arbeidsvaner med programmering

I dette avsnittet skal jeg vise hva studentene uttrykte om arbeidsvanene når de programmerte.

Noe som er veldig tydelig blant studentene er at de ikke jobber med programmering på egenhånd. Dette er noe som kommer frem i både gruppe 2 og 3.

FG3:
«H: (...) vi kan jo ofte velge om vi vil gå etter livekodingen eller ikke, men jeg blir jo heller der og gjør oppgavene, sånn at man slipper å gjøre det hjemme.

J: Samtidig så har man noen å gjøre det med.

H: Ja, det er jo ofte man trenger hjelp.

I: Mhm.

K: Det er litt vanskelig å sitte å programmere alene.»

Det virker som studentene ikke jobber så mye med programmering på egenhånd, fordi de opplever det som utfordrende å programmere alene. Inntrykket jeg sitter igjen med fra studentene er at når de først ikke får til en programmeringsoppgave, syntes de det er vanskelig å løse oppgaven uten hjelp. Dette kan derfor være en forklaring på hvorfor de ikke jobber mye med programmering på egenhånd.

FG3:
«I: Jeg føler at hvis man liksom ikke får det til så er det ikke så lett å gjøre noe med det da.»

Studentene i gruppe 1 forteller at de ofte opplever at oppgavene de må gjøre på egenhånd er mye vanskeligere enn de oppgavene de gjør i gruppetimene.

«B: Ja, og noe av det jeg syntes har vært vanskeligst, er at en ting er det vi går igjennom i timene for det skjer jo på et så grunnleggende nivå, vi har jo lister med fire forskjellige frukter, og så kommer vi hjem også skal vi gjøre oblig der hvor nivået ligger her oppe da. Jeg syntes det er ett ganske stort spenn der. At det da blir vanskelig.»

Forskjellen på hva studentene arbeider med i gruppetimene og det de må gjøre på egenhånd kan også være en forklaring på hvorfor de syntes det er vanskelig å gjøre ting utenfor gruppetimene.

Studentene setter derimot veldig stor pris på å arbeide i gruppetimene og livekodingen. De forteller at de syntes livekodingen er veldig bra, men at de også setter pris på å kunne få hjelp av gruppelærer og medstudenter når de løser oppgaver.

FG3:
«J: Du får sitte med fem andre og det er livekoding.

H: Og de kan hjelpe deg og de viser ting.»

FG1:
«B: Eh, vi sitter og gjør det sammen og så går det veldig sakte og det er en fordel syntes jeg i hvert fall.

A: Ja, og jeg syntes de forklarer veldig mye mer steg for steg.

B: Ja. Også de har jo en mindre mengde studenter å ta hensyn til, så da kan de jo spørre, er alle med.»

Det virker som at studentene gjør seg veldig avhengig av å kunne jobbe i grupper. Sitatene viser at studentene både lærer og jobber med programmeringsoppgaver i grupper. Samtidig forteller flere av studentene at hvis de står fast på en oppgave, så går de raskt og spør om hjelp.

FG3:
«M: Litt sånn generelt, hva er det første dere gjør når dere begynner å løse en programmeringsoppgave? Hvis dere får en oppgave og skal løse den.

J: Spør en venn»

FG2:
M: Bruker dere noen bestemte strategier eller metoder?

C: For koding?

M: Ja?

E: Jeg bruker F.

C: Alle bruker F.

M: Hva gjør F da?

F: Jeg bruker G.

C: Så vi bruker G. Nei, men F og G har veldig peiling.»

Sitatene viser at studentene er flinke til å bruke hverandre og at dette ofte blir noe av det første de tyr til når de skal løse oppgaver. Det virker som at det er en sammenheng mellom studentenes tendens å jobbe i grupper og deres motvilje mot å jobbe på egenhånd. Studentene lærer seg å jobbe med programmering i grupper og gjør seg nesten avhengig av å få hjelp av medstudenter når de skal løse oppgaver.

5 Diskusjon

5.1 Studentenes utfordringer med programmering

Under dette avsnittet vil jeg drøfte det første forskningsspørsmålet:

Hvilke utfordringer møter studentene når de programmerer?

Resultatene fra analysen viser at studentene møter mange av de samme utfordringene som oppstår i andre introduksjonsprogrammer i programmering. Blant dem ligger utfordringer knyttet til å løse store, komplekse oppgaver, forstå hva som skal settes inn i et program, huske og anvende ulike programstrukturer i programmeringen og matematikk.

5.1.1 Problemnedbrytning – Store oppgaver, store programmer

Under avsnitt 4.3.1 identifiserer studentene utfordringer som kan knyttes til problemnedbrytning i modellen til Shute *et al.* (2017). Fokusgruppe 1 og 2 forteller at det er vanskelig å sette sammen store programmer og få de ulike delene av programmet til å jobbe sammen. I følge Liskov og Guttag (2001) innebærer problemnedbrytning å bryte et stort problem ned i mindre delproblemer. Delproblemene må kunne løses hver for seg og når løsningene settes sammen skal de løse det store, originale problemet. Med utgangspunkt i studien til Liskov og Guttag, ser jeg to mulige årsaker til studentenes utfordringer med å lage store programmer. Den ene er at når studentene prøver å bryte ned problemet, sliter de med å identifisere de ulike delproblemene og sette dem sammen. Tilsvarende observasjoner er gjort i andre studier på nybegynnere i programmering (Keen & Mammen, 2015; Raadt *et al.*, 2004). Om dette er tilfellet, blir konsekvensen da at programmet de skriver enten er feil, eller ikke virker når løsningene på hvert delproblem settes sammen til et helt program.

Den andre årsaken kan være at studentene ikke bruker problemnedbrytning som en strategi i det hele tatt. Fra spørreskjemaet forteller en av respondentene at det er vanskelig å starte å programmere når oppgaven er veldig stor. Hvis studentene ikke behersker problemnedbrytning kan det være vanskelig for dem å vite hvordan de skal starte å løse store oppgaver (Keen & Mammen, 2015). I fokusgruppeintervjuene viser ikke studentene til noen form for bevissthet rundt å bryte ned oppgaven. Heller ikke i gruppeoppgavene bruker de

noen form for strategi for å bryte ned oppgaven i mindre deler. Det er ofte sånn at studenter i introduksjonskurs ikke får undervisning i hvordan de skal bryter ned større problemer (Keen & Mammen, 2015). Når vi her ser at studentene hverken nevner noen form for problemnedbrytningsstrategi, eller anvender det i oppgavene sine, da indikerer det at studentene ikke har fått tilstrekkelig undervisning om dette. Jeg mener at studentenes utfordringer med å løse store problemer og å lage store programmer er en kombinasjon av de to årsakene: at de enten deler opp problemet i mindre deler, men sliter med å sette sammen løsningene, eller at de ikke anvender problemnedbrytningsstrategier i det hele tatt. Min studie er ikke nok til å bekrefte om det er den ene eller andre årsaken som er grunnen til denne utfordringen. Derimot er det tydelig at studentene har utfordringer når de skal arbeide med store oppgaver og at de i liten grad anvender problemnedbrytningsstrategier.

5.1.2 Abstraksjon – Å forstå hva som skal programmeres

Fokusgruppene forteller også at det er utfordrende å jobbe med oppgaver med mye tekst. De begrunner dette med at det er så mye informasjon å forholde seg til på en gang. Studien gjort av Haberman og Muller (2008) viste at det var en korrelasjon mellom studentenes manglende evne til å ignorere irrelevante informasjonen og deres utfordringer med å lage løsninger til et problem. Om vi antar at studentenes utfordringer med mye tekst skyldes at de sliter med å skille relevant fra irrelevant informasjon, ser vi likheter mellom studentene i BIOS1100 og studentene i studien til Haberman og Muller. Muller (2005) fant i sin studie at studenter ofte blir distrahert av overfladisk informasjon i problembeskrivelsen. Når fokusgruppene forteller at de syntes oppgaver med mye tekst er utfordrende, kan dette forklares med at de henger seg opp i all den overfladiske informasjonen i oppgaveteksten. I modellen til Shute *et al.* (2017) er abstraksjon beskrevet som en prosess der man ekstraherer essensen av et komplekst system og ignorerer irrelevant informasjon. Dette arbeidet involverer å analysere dataene, lete etter mønster og modellere et system som kan programmeres. Siden abstraksjon handler om å ignorere irrelevant informasjon, er det mulig at studentene hadde opplevde store tekstoppgaver mindre utfordrende om de hadde vært flinkere til å anvende abstraksjon når de programmerer.

Under avsnitt 4.3.2 forteller fokusgruppe 2 og 3 at de ofte syntes det er utfordrende å forstå hva programmeringsoppgavene vil de skal gjøre. En av respondentene fra spørreskjemaet forteller det litt mer konkret og forklarer at det vanskelig å forstå hva oppgaven ber dem

programmere. Utfordringen med å forstå oppgaven er også observert i andre studier (Muller, 2005; Robins *et al.*, 2010). For at studentene skal forstå hva oppgaven ber dem om, må de også ha tilstrekkelige kunnskaper til å forstå hva slags program de blir bedt om å lage. Består oppgaven av en biovitenskapelig problemstilling må studentene i tillegg ha nok biologikunnskaper til å forstå oppgaven. For at studentene i min studie skulle klare å skrive programmet i oppgave c, måtte de beherske både begreper som *recessiv, genotype* og *fenotype,* og sette opp et krysningsskjema for å angi sannsynligheten for at hvert barn arver hvert allel. Wing (2011) beskriver abstraksjon som den viktigste delen av CT. Jeg vil her argumentere for det samme. Den første delen av programmering, handler om å forstå hva som programmeres. Hvis studentene ikke har tilstrekkelige kunnskaper til å forstå hva oppgaven ber dem om å gjøre, har det liten betydning om de har kunnskapene til å skrive et fungerende program.

Studien gjort av Qualls *et al.* (2011) viste at selv om studentene ikke forstod hva abstraksjon var, hadde de tilstrekkelige kunnskaper til å anvende det når de programmerte. Studentene i fokusgruppe 2 forteller at programmering handler om å ta ut de viktige delene av en oppgave. Dette gir en antydning om at studentene anvender abstraksjon, men at de gjør dette ubevisst. Grunnen til at studentene ikke bevisst anvender abstraksjon har nok en sammenheng med at de ikke får undervisning i det. Det kan også være tilfelle at de får undervisning i abstraksjon, men at de ikke forstår det, noe som er vist i flere studier (Hazzan & Kramer, 2007; Qualls *et al.*, 2011).

5.1.3 Algoritmer – Programstrukturer og matematikk

Studentene forteller i avsnitt 4.3.3 at det er utfordrende å sette sammen et program på riktig måte. Jeg har allerede drøftet noen sider ved dette i avsnitt 5.1.2, men da var det i forhold til store programmer og problemnedbrytning. I dette avsnittet vil jeg drøfte utfordringene studenten opplever når de skal skrive et program i Python. Med utgangspunkt i svarene fra både fokusgruppene og spørreskjemaet, er det å bruke programstrukturer en utfordring. Studentene sliter både med å ha kontroll over alle detaljene som må være med inn i ett program, men også med å forstå hvordan de ulike programstrukturene virker og hvordan de skal plasseres i forhold til hverandre. Denne formen for utfordringer kan knyttes til deres konseptuelle kunnskap (Bayman & Mayer, 1983). Med konseptuell kunnskap menes forståelsen av de ulike strukturene og prinsippene til et program. Studentenes utfordringer

viser tydelige mangler i deres konseptuelle kunnskaper og det virker som at jobben med å skrive programmer blir veldig komplekst for studentene. Utfordringer med programstrukturer er på sin side noe som er vanlig blant studenter i introduksjonskurs (Medeiros *et al.*, 2018). Usikkerheten med å bruke de ulike programstrukturene ser ut til å skape utfordringer når studentene skal starte å programmere. Siden de strategiske kunnskapene bygger på studentenes konseptuelle kunnskaper (McGill & Volet, 1997) er det naturlig at studentene også sliter med å planlegge og skrive programmer. Samtidig kan deres utfordring med å starte programmeringen også henge sammen med de synes det er vanskelig å forstå hva oppgaven dem om å programmere (som diskutert i avsnitt 5.1.2). For at studentene skal klare å lage et program må de både forstå hva oppgaven krever av dem og vite hvordan de faktisk lager programmet.

Et annen parrallell som kan trekkes mellom utfordringene i algoritmer og abstraksjon er at abstraksjon ofte innebærer å arbeide i forskjellige nivåer (Wing, 2011). Ved å arbeide i et høyere nivå i Python, kan man ignorere hvordan de lavere nivåene av programmet virker. Det kan derfor tenkes at når studentene forteller at de sliter med alle detaljene som skal inn i programmet, er dette en konsekvens av at de ikke klarer å forholdet seg til et høyere nivå i abstraksjonen. Hvis de ikke klarer det, ender de opp med å programmere i flere nivåer samtidig, noe som fører til unødvendig mange detaljer og dermed øker kompleksiteten i oppgaven.

Den algoritmiske delen av CT handler om å kunne lage et sett med strukturerte steg for å løse et problem (Shute *et al.*, 2017). Ofte handler dette om å lage en matematisk algoritme som kan skrives som et program. I følge Gomes *et al.* (2006) kan studentene slite med å gjøre et problem om til matematisk form. Studentene i fokusgruppe 3 forteller at de synes det er vanskelig å lage en egen likning når de programmerer. De forteller også at det er vanskelig å skrive algoritmen på en form som programmet deres oppfatter. Det er altså ikke bare det å skrive en algoritme som er utfordrende, men også å lage et program av algoritmen. Matematikk trekkes frem som en stor utfordring i alle fokusgruppene og det utgjør den mest omtalte utfordringen i spørreskjemaet. Berg (2019) observerte også dette i sin studie, der hun så at matematikk ble oppfattet som mest utfordrende i BIOS1100. Når matematikkunnskapene ikke er tilstrekkelige, vil studentene slite med å lage algoritmene sine (Gomes & Mendes, 2010). Matematikken ser derfor ut til å være en krevende faktor for studentene når de programmerer.

Fokusgruppe 1 og 3 forteller at de sliter med å forstå matematikken som introduseres i BIOS1100. De opplever at matematikken ligger på et nivå som forutsetter at de har hatt R2 matematikk på videregående. En av studentene som har hatt R2 forteller derimot at matematikken i BIOS1100 fortsatt oppleves veldig krevende og på et høyere nivå enn hva studenten forstår. Interessant nok viser også studien til Berg (2019) at rundt 1/3 av studentene på BIOS1100 har hatt R2 matematikk, men at det ikke har vært en fordel for de studentene. Studentenes forkunnskaper i matematikk virker å være en utfordring når de lærer programmering, selv for studentene med R2 bakgrunn fra videregående. At studentene ikke har tilstrekkelige forkunnskaper i matematikk er også sett i andre studier (Gomes & Mendes, 2010). Man kan stille spørsmål ved hvorvidt studentenes bakgrunn med matematikk er for dårlig, eller om innholdet i BIOS1100 ligger på et nivå som er høyere enn hva studentene har forutsetninger for å forstå. Fra høsten 2019 er det krav om at alle som tar BIOS1100 har R2 matematikk. Det vil derfor være interessant å se om dette har påvirkning på studentenes utfordringer med matematikk i emnet.

5.1.4 Feilsøking - Ulike feil i programmene

Under avsnitt 4.3.4 forteller studentene om ulike feil de vanligvis gjør når de programmerer. Det ser ut som at den vanligste feilen studentene gjør når de programmerer er syntaksfeil. Flere studier viser nettopp at studenter ofte gjør syntaksfeil når de starter å programmere (Altadmri & Brown, 2015; Medeiros *et al.,* 2018). Studenten forteller at de vanligvis får feil fordi de enten mangler kolon, eller har en skrivefeil. Studentene i fokusgruppe 2 trekker også frem at de gjør konseptuelle feil (Bayman & Mayer, 1988), som å plassere *choice* funksjoner feil i programmet. Det inntrykket jeg sitter igjen med fra intervjuene er at studentene er veldig komfortable med å løse både syntaks- og konseptuelle feil. Syntaksfeil er som regel ikke noe studentene bruker mye tid på å reparere (Altadmri & Brown, 2015), men konseptuelle feil kan ofte være utfordrende for dem (Bayman & Mayer, 1988). Her ser vi altså forskjell i hva studentene i min studie sier og hva studien til Bayman og Mayer (1988) viser. Samtidig har ikke mine intervjuer gått i dybden av de ulike feilene studentene gjør, så det kan godt være at de opplever større utfordringer med konseptuelle feil enn hva sitatene viser. Et par av studentene i fokusgruppe 2 forteller også at de av og til får feil svar på oppgaven, selv om programmet de lager virker som det skal. Dette er et eksempel på logiske feil (Hristova *et al.*, 2003). Studentene gir ingen forklaring på hva som er opphavet til de logiske feilene, så det er vanskelig å si om det skylds algoritmiske feil, feiltolkning av oppgaven eller misoppfatninger

(Ettles *et al.*, 2018). Allikevel har jeg i de tre siste avsnittene redegjort for at studentene synes det er utfordrende å forstå hva oppgaven ber dem om, lage algoritmer og skrive store programmer. Det er derfor ikke urimelig å anta at de logiske feilene oppstår som en konsekvens av studentenes utfordringer med dette.

5.2 Studentenes strategier for programmering

Her vil jeg drøfte det andre forskningsspørsmålet i min studie:

Hvilke strategier bruker studentene når de programmerer?

Resultatene fra dette avsnittet viser at studentene har en del utilstrekkelige eller fraværende programmerings- og problemløsningsstrategier. Deres arbeid er i stor grad knyttet til å anvende eksempelprogrammer og prøv- og feil-strategier.

5.2.1 Abstraksjon – Analyse av oppgaven

Det første trinnet i Pólyas definisjon av problemløsning er å forstå hva problemet er (Pólya, 1945). I arbeidet med CT er det problemnedbrytning og abstraksjon som er spesielt viktig for å forstå hva problemet er (Shute *et al.*, 2017). Som jeg allerede har drøftet i avsnitt 5.1.1 virker det som at studentene ikke bruker bevisste strategier for å bryte ned store oppgaver, noe som indikerer ineffektiv eller fraværende bruk av problemnedbrytning. Når det kommer til abstraksjon involverer dette at studentene klarer å skille ut den essensielle informasjonen for oppgaven ved å identifisere og analysere informasjonen (Shute *et al.*, 2017). I to av fokusgruppene forteller studentene at de leser gjennom oppgaveteksten flere ganger for å finne ut hva oppgaven spør om. Studentene i den tredje fokusgruppa forteller at de prøver å lage en oversikt over all informasjonen og sirkle ut det de anser som viktig for oppgaven. Disse resultatene gir ikke ett bilde av studenter med mange strategier for å finne den essensielle informasjonen fra oppgavene. Å lese teksten flere ganger kan være et nyttig verktøy for å sette seg inn i tekstens innhold, men det kan nok ikke regnes som en effektiv strategi for å skille relevant fra irrelevant informasjon. Den tredje fokusgruppen viser derimot til en mer effektiv strategi for å finne den relevante informasjonen, men de knytter den opp mot det å lage krysningsskjemaer i biologien. Det er diskutabelt om denne strategier først og fremt er noe de bruker når de arbeider med biologioppgaver eller om de også anvender denne formen for modellering når de programmerer. Studenter som lærer programmering har ofte

utfordringer med å analysere problemer og dette skyldes ofte at introduksjonskursene ikke lærer dem problemløsningsstrategier (Hazzan *et al.*, 2011). Som jeg drøftet i avsnitt 5.1.2 har studentene utfordringer med å løse store tekstoppgaver. Det virker som at dette har en sammenheng med deres manglede strategier for å finne den relevante informasjonen og deres mangelfulle strategiske kunnskaper.

5.2.2 Algoritmer – Prøving- og feiling-strategier

Når jeg spør studentene om hvordan de programmerer svarer alle tre fokusgruppene at de prøver seg frem. Sørby og Angell (2012) fant i sin studie at også norske fysikkstudenter benytter seg av mye prøving og feiling for å løse programmeringsoppgaver. Deres argument er at studentene bør få mer oppfølging, slik at de får et mer bevisst forhold til strategiene de benytter seg av. Prøv og feil-strategier trenger ikke å være negativt i seg selv, men det er grunn til å være skeptisk når dette blir en av få strategier studentene anvender. De utfordringene og strategiene som jeg har presentert i denne studien tyder på at studentene har utilstrekkelige problemløsnings- og programmeringsstrategier Dette samsvarer med andre studier hvor det er vist at studenter ofte har utilstrekkelige strategier for å løse programmeringsoppgaver (Clancy & Linn, 1999; Lister *et al.*, 2006). Dette kan ha en sammenheng med at studentene har dårlige problemløsningsstrategier fra før (Medeiros *et al.*, 2018). Samtidig kan det skyldes at det ikke er tilstrekkelig fokus på programmering- og problemløsningsstrategier i introduksjonskursene. Hazzan *et al.* (2011) mener at prøv- og feil-strategier oppstår som følge av manglende undervisning av problemløsningsstrategier. Med bakgrunn i mine funn, og studien til Sørbye og Angell (2012) og Hazzen *et al.* (2011) er det mye som tyder på studentenes bruk av prøv- og feil-strategier er et resultat av manglende strategisk kunnskap hos studentene. Siden jeg har sett i denne studien at studentene sliter med å planlegge og skrive programmer, mener jeg at prøving- og feiling som strategi er en dårlig erstatning for mer strukturerte problemløsnings- og programmeringsstrategier.

Et annet funn er at studentene anser prøving og feiling som en normal arbeidsform i programmeringen, men ikke i biologi. Dette er interessant fordi det viser at studentene arbeider med biologi og programmering forskjellig. En grunn til dette kan være at mens biologi typisk er et fag med ett riktig svar, kan man svare på en programmeringsoppgave med mange ulike løsninger. En av studentene uttrykker også frustrasjon over nettopp det at de har

muligheten til å lage så mange forskjellige løsninger. Det er derfor mulig at biologien oppleves som mer trygt fordi studentene vet at det kun er ett riktig svar.

5.2.3 Feilsøking – Identifiser og reparer

Under avsnitt 4.3.4 forteller fokusgruppene at de anvender de samme metodene for å feilsøke programmene. Studentene forklarer at når de får opp en feilmelding prøver de først å identifisere feilen. Når de har identifisert feilen går de rett inn i programmet og forsøke å reparere den. Denne formen for feilsøking er den samme Kummerfeld og Kay (2003) identifiserte hos nybegynnere i sin studie. De mener at denne måten å feilsøke på fungerer fint så lenge det kun er syntaksfeil i programmet, men ikke hvis programmet i seg selv er feil. Dette samsvarer med det jeg diskuterer i avsnitt 5.1.4, der vi ser at studentene er komfortable med syntaks- og konseptuelle feil, men synes det kan være utfordrende med logiske feil (Bayman & Mayer, 1988). Deres utfordringer med de logiske feilene kan skyldes at studentene feilsøker som de gjør. Kummerfeld og Kay (2003) identifiserte at erfarne programmerere også vurderer hele programmet, ikke bare identifiserer feilen og reparerer den. Om studentene hadde anvendt dette i sin egen feilsøking, kan det tenkes at det hadde vært lettere for dem å korrigere de logiske feilene. Samtidig har jeg ikke nok data til å hverken bekrefte eller avkrefte dette, men fra observasjonene jeg har gjort, ser det ikke ut som at feilsøking er studentenes største utfordring med programmering.

Et annet verktøy studentene bruker for å kontrollere at programmet er riktig er at de kjører programmet flere ganger. Studentene i fokusgruppe 2 mener dette er en viktig del av programmeringen og viser misnøye med at de ikke kan kjøre programmene sine på eksamen. I BIOS1100 lærer studentene programmering gjennom å kjøre programmene sine, men når de skal vurderes til eksamen har de ikke den muligheten. Ifølge Biggs og Tang (2011) må det være sammenheng mellom læringsformen til studentene og sluttvurderingen. Dette kaller man for samstemt undervisning. For å oppnå samstemt undervisning må man blant annet ha vurderingsformer som gjør det mulig å vurdere studentenes prestasjon. Videre må læringsaktivitetene føre til at studentene når læringsmålene. At studentene lærer seg å kjøre programmene, men ikke får bruke dem på eksamen, er et tegn på at det ikke er samstemt undervisning i BIOS1100. Derfor kunne det vært hensiktsmessig å åpne opp for at studentene kan kjøre programmene sine på eksamen. Samtidig forklarer studentene i fokusgruppe 1 at de foretrekker å ikke kunne kjøre programmet på eksamen, fordi dette kan føre til at de låser seg

fast i oppgaven om programmet ikke virket. Det hadde vært interessant å se om det å kjøre programmet hadde hatt betydning for studentenes resultater på eksamen. Forutsetter man at studentene har de nødvendige konseptuelle- og syntaktiske ferdighetene, bør de kunne skrive et fungerende program uten å måtte kjøre den.

5.2.4 Bruk av eksempelprogrammer

En tydelig trend hos alle fokusgruppene er at de bruker ferdige programmer for å løse nye programmeringsoppgaver. Studentene forteller at de enten ser på programmer de selv har laget eller som er laget av gruppelærer i livekodingen. Observasjonene jeg gjorde av gruppenes arbeid med oppgavene viste at studentene brukte tidligere oppgaver når de skulle lage programmet i oppgave c. Her bruker jeg definisjonen til Müller *et al.* (2018) og definerer de ferdige programmene som eksempelprogrammer. Studien gjort av Müller *et al.* (2018) viser at det er ulike grunner til at studentene bruker eksempelprogrammer. Det jeg ser i min studie er at studentene først og fremst ser på eksempelprogrammer for å forstå og huske hvordan de skal anvende ulike programstrukturer i det nye programmet. Her ser det ut som at studentene bruker eksempelprogrammer for å forstå programmeringsspråket, noe som samsvarer med en av de observasjonene til Müller *et al.* To av gruppene startet også oppgave c med å se etter tidligere programmer som liknet på oppgaven. Da ser det ut som at de bruker eksempelprogrammene for å forstå oppgaven bedre, noe Müller *et al.* også observert i sin studie. Selv om det ikke snakkes mye om i intervjuene, observerte jeg at studentene brukte deler av eller hele eksempelprogrammet for å forstå hvordan de skulle skrive sitt eget program. Indirekte forteller fokusgruppene at de først og fremst ser på tidligere oppgaver som en hjelp for å lage programstrukturer i programmet. Müller *et al.* identifiserte tilsvarende bruk av eksempelprogrammer i sin egne studie og omtaler det som at studentene bruker eksempelprogrammer som en referanse for sitt eget program.

Neal (1989) mener at bruk av eksempelprogrammer er viktig for både nybegynnere og erfarne programmerere. Studentene i min studie viser konsekvent bruk av eksempelprogrammer når de programmerer, og i lys av Neals synspunkter er det positivt at studentene anvender seg av denne strategien. Gaspar og Langevin (2007) er dermed skeptiske til det de definerer som «klipp og lim» programmering og stiller spørsmål ved hvorvidt dette fostrer læring av programmering. Sitatene fra fokusgruppe 3 gir inntrykk av at selv om de klarte å lage et program for oppgave c, har de ikke forstått hvordan de lagde den. Fokusgruppe 1 forteller at

de tror det hadde vært vanskelig å lage programmet i oppgave c uten hjelpemidler. Dette kan bety at det å bruke eksempelprogrammer også har en bakside. Hvis det blir en dominerende strategi kan det føre til at studentene ikke lærer seg å programmere skikkelig, og dermed får problemer med å skrive programmer uten hjelpemidler. Resultatet blir studenter som klarer å skrive et program, men ikke forstår hvordan programmet faktisk virker. Siden studentene forteller at de ofte kombinerer eksempelprogrammer med prøving og feiling (som drøftet i 5.2.2), ser det ut som at dette blir en erstatning for manglende strategiske kunnskaper hos studentene. Siden de heller ikke kan bruke hjelpemidler på eksamen i BIOS1100, er dette igjen et eksempel på at det ikke er helt samsvar mellom hvordan studentene lærer seg å programmere og hvordan de vurderes i emnet. Dette er uheldig, da man ønsker at læringsaktivitetene samsvarer med læringsmålene (Biggs & Tang, 2011).

5.3 Rollen til det biofaglige i programmering

Dette avsnittet ser på hvordan studentene arbeider med programmering og biologi sammen, og svarer på følgende forskningsspørsmål:

«Hvilken rolle spiller det biofaglige når studentene programmerer?»

Resultatene fra analysen viser at biologiske problemstillinger ikke påvirker studentenes arbeid med å skrive selve programmet. Derimot virker det som at biologiske problemstillinger har en positiv effekt for studentens interesse for å programmere, men at det kan gjøre det vanskeligere å løse programmeringsoppgaver.

Det første jeg ser her er studentenes arbeid med å forstå hva oppgaven ber dem om. Jeg har allerede drøftet en del av dette i avsnitt 5.1.2, der studentene forteller at syntes det er vanskelig å forstå hva oppgaven ber dem om å gjøre. To av studentene i fokusgruppe 2 forteller at de synes det er enklere å programmere uten biologiske problemstillinger. Guzdial (2010) påpeker at det kan være en utfordring for studentene å programmere i en kontekst og mener at spesielt svake studenter kan bli distrahert av all informasjonen de må forholde seg til. Samtidig er resten av studentene i fokusgruppene positive til å programmere med biologiske problemstillinger og de forklarer at den biologiske informasjonen gir dem en pekepinn på hva svaret blir. Til tross for at studentene har litt forskjellige meninger om oppgaver med biologiske problemstillinger, trenger ikke den ene å ekskludere den andre. Oppgaver med biologisk informasjon kan ha en positiv effekt på studentenes evne til å forutse

hva programmet skal lage, men samtidig kan det gjøre oppgaven vanskeligere hvis studentene ikke har nok kunnskaper til å løse den biologifaglige delen.

Under avsnitt 4.3.6 snakker studentene om den biologifaglige påvirkningen på å skrive programmer. Her kommentere studentene i to av fokusgruppene at det biologifaglige ikke har noe å si når de programmerer. Bouvier *et al.* (2016) har også tilsvarende funn i deres studie, der de så at programmering i en kontekst ikke hadde betydning på studentenes problemløsning. En av studentene i min studie påpeker at så lenge de vet hva de skal programmere, har ikke det biofaglige noe å si for selve programmeringen. Tilsvarende har man sett i andre studier som har undersøkt programmeringskurs i biologi. Der har resultatene vært at studentene i programmeringskurs i biologi er like flinke til å skrive programmer som studenter i andre programmeringskurs (Berger-Wolf *et al.*, 2018; Dodds *et al.*, 2012). Det ser derfor ikke ut som at biologi hverken har en positiv eller negativ effekt på studentenes arbeid med å programmere.

Det er litt uenighet blant studentene om hvorvidt det biologifaglige gjør programmering mer interessant. Alle bortsett fra to studenter svarer at de er mer positive til å programmere når dette gjøres i biologifaglige problemstillinger. Fra sitatene ser det ut som at studentene tar programmeringen fordi de må, men at de er mer positive til programmering når det er sammen med biologi. Dette samsvarer med andre funn, som ser at det å programmere i en kontekst kan gjøre studentene mer motivert for programmeringen (Berger-Wolf *et al.*, 2018; Forte & Guzdial, 2005). Samtidig svarer også to av fokusgruppene at de ser relevansen av å lære seg programmering, og at de ser på det som relevant for fremtiden.

En interessant observasjon var at de to studentene som heller ville ha et rent programmeringskurs, var gutter. Totalt var det 3 gutter av de 11 studentene som deltok i denne studien. Dette kan ikke generaliseres, men det gir et inntrykk av at guttene finner programmeringen mer interessant enn jentene. Berg (2019) fant også at guttene i BIOS1100 var litt mer interessert i emnet enn jentene. Siden BIOS1100 er et programmeringsemne, kan man anta at det er selve programmeringen som gjør emnet mer interessant for guttene. De to guttene i fokusgruppeintervjuet svarte også at de hadde hatt programmering før de kom til UiO. På spørsmålet om hva studentene følte de mestret i BIOS1100, var det kun de to guttene som svarte programmering. Resten av studentene svarte biologi. Også her ser studien til Berg (2019) tilsvarende funn, der guttene hadde litt større forventing til å mestre BIOS1100 enn jentene. Dette støtter også studien til Hagan og Markham (2000) som så at studenter med

programmeringserfaring ofte har bedre programmeringsferdigheter og større selvtillit i programmering enn studenter som ikke har det

I dette siste avsnittet vil jeg kort drøfte en observasjon fra studentenes arbeid med oppgavesettet. Når studentene arbeidet med oppgavene så jeg at de arbeidet med et programmeringsrettet-fokus. I oppgave a var gruppe 1 og 2 usikre på hva svaret ble, fordi de så etter en måte å programmere oppgaven på. Svaret på oppgaven (se vedlegg D) var at besteforeldrene enten kunne være homozygot dominant, eller heterozygot. Studentene lette derimot etter én riktig genotype, fordi de tenkte at svaret skulle inn i et program. Tilsvarende så jeg også i oppgave b, der alle studentene fikk feil svar, noe som kan skyldes at de enten misforstod oppgaven, eller ikke anvendte de biologiske kunnskapene skikkelig for å se hva oppgaven spør etter. Både situasjonen i oppgave a og b viser at studentene sliter med å kombinere kunnskapene innenfor programmering, matematikk og biologi. Sørby og Angell (2012) observerte også dette hos sine studenter.

5.4 Studentenes arbeidsvaner med programmering

Her vil jeg drøfte det siste forskningsspørsmålet:

«Hvilke arbeidsvaner har studentene med programmeringsoppgaver i emnet BIOS1100?»

Studentenes arbeidsvaner er et relativt stort tema og jeg har kun sett en liten del av dette min studie. Det er allikevel én observasjon jeg har gjort som er interessant. Studentene forteller at de arbeider lite med programmering på egenhånd. Det ser ut som at dette skyldes at de syntes det er vanskelig å programmere alene. Blant annet forteller studentene at hvis de står fast i en oppgave, får de ikke gjort så mye på egenhånd. Dette var også en av observasjonene gjort av Kummerfeld og Kay (2003). De så at når nybegynnere programmerte spurte de ofte om hjelp, eller brukte andre hjelpemidler for å komme seg videre. Studentene forteller også at de syntes oppgavene de arbeider med på egenhånd er vanskeligere enn hva de arbeider med i gruppetimene. Videre forteller studentene at de er positive til livekodingen i BIOS1100 og at de setter pris på å kunne jobbe i grupper. Til tross for dette, tolker jeg det som at studentene blir litt for avhengig av gruppene de arbeider i. Dette kan man se når studentene forklarer at de gjerne starter en programmeringsoppgave med å spørre om hjelp fra andre medstudenter.

Bruk av gruppearbeid i undervisningen har vist positiv effekt på både læring av modeller i fysikk (Sørby & Angell, 2012) men også på elevenes sluttvurdering i programmeringskurs (Beck *et al.*, 2005). Derfor kan det tenkes at studentene har et godt faglig utbytte av å jobbe i gruppene på BIOS1100.

Et annet konsept ved gruppearbeidet i BIOS1100 er at studentene ofte løser oppgaver på én felles datamaskin. Med mindre dette reguleres av studentene, vil det involvere at det er én student som sitter og programmerer hele tiden. Da er det fort at noen studenter blir passive og ikke lærer like mye. Samtidig samsvarer utforming av gruppearbeidet i BIOS1100 godt med de faktorene som LeJeune (2003) anser som kritiske for samarbeidende læring. Gruppene er små og de arbeider med felles oppgaver i gruppa. Hvorvidt studentene har en samarbeidende adferd er ikke noe jeg kan svare på gjennom min studie. Studentene er i alle fall positive til å arbeide i gruppene og fra forskningens side er det mange positive sider med gruppearbeid.

5.5 Implikasjoner for undervisningen av programmering

5.5.1 Implikasjoner for BIOS1100

Siden denne studien har sett på hvordan studentene i BIOS1100 arbeider har funnene stor betydning for undervisningen der. Den største observasjonen fra denne studien er at studentene har nokså mangelfulle programmerings- og problemløsningsstrategier. Derfor mener jeg det vil være fordelaktig om dette integreres i undervisningen ved senere gjennomføringer av emnet. En studie har vist at nettopp CT kan ha positiv effekt på studentenes evne til problemløsning i programmering (Fernández *et al.*, 2018). Siden CT er ment å være en effektiv måte å løse problemer på (Wing, 2006) ville det vært en fordel om dette ble undervist i BIOS1100. CT kan undervises gjennom både problemløsning og samarbeid Hsu *et al.* (2018) og BIOS1100 anvender allerede disse undervisningsformene i sitt kurs. Den største tilpasningen BIOS1100 måtte ha gjort, ville da ha vært å introdusere de ulike komponentene av CT (Shute *et al.*, 2017). Her har andre studier vist at ved å for eksempel settes fokus på aktiviteter som krever at studentene må anvende problemnedbrytning, blir studentene flinkere til dette når de programmerer (Muller *et al.*, 2007; Pearce *et al.*, 2015). Ved å introdusere hva CT er, og tilrettelegge for at studentene

lærer å bruke det, er det mye som tyder på at det kan ha en positiv effekt på studentens strategiske kunnskaper når de programmerer.

Det bør også tas hensyn til forholdet mellom undervisningsformer og eksamen (Biggs og Tang, 2011). Siden eksamen i BIOS1100 ikke gir mulighet for å kjøre programmet som studentene lager, bør de trenes på å kunne lage programmer uten å måtte teste dem. Dette innebærer at det tilrettelegges for en dypere forståelse av hvordan programmering fungerer hos studentene. Videre bør det også tas hensyn til at eksamen er en individuell eksamen. Som vist i min studie så jobber studentene hovedsakelig i grupper. For at studentene skal være bedrer rustet til å programmere på egenhånd, bør det legges opp til at de jobber mer på egenhånd gjennom undervisningen i BIOS1100.

Jeg tror også det er hensiktsmessig for emneansvarlige i BIOS1100 å ha et reflektert forhold hvordan studentene bruker eksempelprogrammer i emnet. Bruk av eksempelprogrammer er ifølge Neal (1989) en viktig strategi for å programmerer, men som jeg har vist i min studie er dette en strategi som fort dominerer studentenes programmering. Hvis man ønsker studenter som programmerer selvstendig, er det viktig at de har en mer omfattende verktøykasse av strategier når de programmerer.

5.5.2 Implikasjoner for implementering av programmering i realfaglige emner

Mye av det jeg har presentert i forrige avsnitt kan også relateres til programmering i andre realfaglige emner. Mange av implikasjonen for BIOS1100 kan selvfølgelig ikke relateres til andre emner, ettersom min studie er gjort i den undervisningsformen som BIOS1100 har. Eksempelvis bruker ikke alle emner livekoding som et verktøy, eller anvender gruppearbeid i like stor grad. Utenom det ser vi at de utfordringene som studentene i BIOS1100 møter tilsvarer observasjoner fra andre studier, så det er tydelig at dette er noe som går igjen i programmeringskurs. Igjen anser jeg studentens mangler på problem- og programmeringsstrategier som særlig utfordrende for studentene. Dette er noe jeg mener det bør legges større vekt på når man lærer studenter å programmere.

5.6 Begrensninger av studien og forslag til videre forskning

I denne studien har jeg sett på et lite utvalg studenter mens de programmerer i BIOS1100. For å kunne generalisere funnene er jeg avhengig av tilfeldig valgte informanter, noe jeg ikke får når jeg selektivt velger studenter fra BIOS1100. Utvalget mitt er heller ikke stort nok til at jeg kan generalisere funnene til å gjelde andre programmeringskurs. Min studie har generelt vært meget deskriptiv. Jeg har forsøkt å identifisere hvordan studentene arbeider med programmering i biologi, og jeg har forsøkt å se på dette gjennom CT. Det at oppgaven har vært såpass vid, betyr at jeg ikke har fått gått ordentlig i dybden av de ulike strategiene, utfordringene og arbeidsvanene til studentene. Derfor tror jeg det er mye forskning som kan gjøres for å undersøke studentens arbeid med de seks CT-komponentene til Shute *et al.* (2017). Det at jeg ikke filmet eller tok bilde av programmene som ble laget av studentene fører også til at jeg ikke kan si mye om programmene de laget. Her er det også mange interessante temaer som kan undersøkes, om dette gjøres i fremtidige studier.

Denne studien er gjort på et fagfelt som er relativt ungt og jeg håper at min studie blir et bidrag for videre utvikling. Jeg har derfor følgende forslag til videre studier:

- *R2 og mattekrav:* Fra høsten 2019 er det krav om at alle som skal studere biovitenskap ved UiO har R2 matte eller tilsvarende. Hvordan vil dette påvirke studentenes ferdigheter og utfordringer når de skal programmere?

- *Effekten av strategier*: Hvordan påvirkes studentenes arbeid med programmering hvis det legges større vekt på å undervise problemløsningsstrategier?

- *Effekten av CT*: Hvordan påvirkes studentenes arbeid med programmering hvis de introduseres for CT i undervisningen?

- *Biologi og programmering:* Hvordan påvirkes studentens arbeid med programmering hvis du gir dem åpne, problembaserte biologioppgaver?

6 Konklusjon

I denne studien har jeg forsøkt å undersøke hvordan studenter arbeider med programmeringsoppgaver i biovitenskapelige problemstillinger. Siden dette er et stort spørsmål har jeg forsøkt å svare på dette gjennom fire forskningsspørsmål. Jeg vil nå gi en kort oppsummering av funnene jeg har i denne studien.

Det første spørsmålet omhandlet hvilke utfordringer studentene møtte når de programmerte. De funnene jeg har viser at studentene opplever de samme typer utfordringer som man ser i andre introduksjonskurs i programmering. Studentene har vist at det er utfordrende å skrive store programmer og at de sliter med å finne ut hva programmeringsoppgavene krever av dem. Arbeidet med å skrive selve programmet har også vist seg å by på flere utfordringer hos studentene. I stor grad handler dette om at studentene mangler konseptuelle kunnskaper, noe som fører til at de er usikre på hvilke programstrukturer de skal bruke, hvordan strukturene virker og hvordan de setter sammen hele programmet. Videre byr matematikken på utfordringer hos studentene. Her opplever studentene at matematikken ligger på et nivå de ikke behersker og deres forkunnskaper i matematikk skaper utfordringer for dem i BIOS1100. Studentene opplever også en del typer feilmeldinger når de programmerer, men det virker ikke som en stor utfordring for dem.

Når det kommer til studentenes strategiske kunnskaper, så virker disse å være nokså svake. Innenfor modellen til Shute *et al.* (2017) ser vi at studentene bruker få strategier når de programmerer. Det er i stor grad to strategier studentene bruker. Den ene er bruk av eksempelprogrammer og den andre er prøv- og feil-strategier. Jeg har i denne oppgaven argumentert for at å bruk av eksempelprogrammer ikke er negativt i seg selv, men at det blir ineffektivt for studentens arbeid når dette blir en dominerende strategi. Begge strategiene virker å være en kompensasjon for mangelen av mer effektive strategier og det er mye som tyder på at studentenes utfordringer henger sammen med deres mangelfulle strategiske kunnskaper.

Det tredje spørsmålet har sett på hvordan biovitenskapelige spørsmål påvirker studentens programmering. Gjennom denne studien har jeg sett at det biologifaglige ikke påvirker studentens arbeid med å skrive selve programmet. Derimot virker det å ha positivt effekt på deres evne til å løse programmeringsoppgaver, siden studentene får en pekepinn på hva svaret

i oppgaven skal bli. Samtidig gjør det oppgaven vanskeligere, fordi det krever at de både må bruke biologi- og programmeringskunnskaper. Bruk av biovitenskapelige oppgaver fører også til at programmeringen blir mer interessant for de fleste studentene.

Til sist har jeg sett på hvordan studentene arbeider i BIOS1100. Her har jeg sett at de nesten ikke arbeider med programmering på egenhånd, men at de jobber mest i grupper. Dette skyldes at det er vanskelig å programmere på egenhånd og jeg anser det derfor som viktig å tilrettelegge for mer individuelt arbeid i emnet.

Programmering begynner å sette sitt merke på den realfaglige utdanningen. På Universitet i Oslo er programmering allerede godt etablert i mange av realfagene og nå blir det også en del av matematikk og naturfag i norsk grunn- og videregående skole. Samtidig er programmering i realfaglig utdanning et ungt fagfelt og det er lite vitenskapelig konsensus om hvordan realfaglig programmering bør undervises. Jeg har i min studie avdekket en rekke utfordringer hos studenter som programmerer i biologi. Noen er generelle for programmering, mens andre kan relateres til det biologifaglige. Mange av utfordringene ser ut til å være en konsekvens av studentenes mangelfulle programmerings- og problemløsningsstrategier. Dette er viktige funn fordi vi som lærere og professorer må ta hensyn til studentenes forutsetninger og utfordringer når vi utdanner dem i programmering.

Litteraturliste

Altadmri, A., & Brown, N. C. C. (2015). *37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data*. Artikkel er presentert i Proceedings of the 46th ACM Technical Symposium on Computer Science Education, Kansas City, Missouri, USA.

ATLAS.ti GmbH. Hentet fra https://atlasti.com/

Barr, D., Harrison, J., & Conery, L. (2011). Computational thinking: A digital age skill for everyone. *Learning & Leading with Technology, 38*(6), 20-23.

Bauer, M. W., & Gaskell, G. (2000). *Qualitative researching with text, image and sound : a practical handbook*. London: Sage.

Bayman, P., & Mayer, R. E. (1983). A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Commun. ACM, 26*(9), 677-679.

Bayman, P., & Mayer, R. E. (1988). Using conceptual models to teach BASIC computer programming. *Journal of Educational Psychology, 80*(3), 291-298.

Beck, L. L., Chizhik, A. W., & McElroy, A. C. (2005). Cooperative learning techniques in CS1: design and experimental evaluation. *SIGCSE Bull., 37*(1), 470-474.

Berg, M. M. (2019). *Studentar si interesse og meistringsforventning for programmering og modellering i biologi: Ein kvantitativ studie av studentar si interesse og meistringsforventning for programmering og modellering i biologi gjennom emnet BIOS1100.* (Masteroppgave), Universitet i Oslo, Oslo.

Berger-Wolf, T., Igic, B., Taylor, C., Sloan, R., & Poretsky, R. (2018). *A Biology-themed Introductory CS Course at a Large, Diverse Public University*. Artikkel er presentert i Proceedings of the 49th ACM Technical Symposium on Computer Science Education.

Berland, M., & Wilensky, U. (2015). Comparing virtual and physical robotics environments for supporting complex systems and computational thinking. *Journal of Science Education and Technology, 24*(5), 628-647.

Biggs, J. B., & Tang, C. (2011). *Teaching for quality learning at university : what the student does* (4th ed. ed.). Berkshire: Society for Research into Higher Education & Open University Press.

Bouvier, D., Lovellette, E., Matta, J., Alshaigy, B., Becker, B. A., Craig, M., . . . Zarb, M. (2016). *Novice Programmers and the Problem Description Effect*. Artikkel er presentert i Proceedings of the 2016 ITiCSE Working Group Reports, Arequipa, Peru.

Braun, V., & Clarke, V. (2006). Using thematic analysis in psychology. *Qualitative research in psychology, 3*(2), 77-101.

Chattopadhyay, S., Nelson, N., Nam, T., Calvert, M., & Sarma, A. (2018). *Context in programming: an investigation of how programmers create context*. Artikkel er presentert i Proceedings of the 11th International Workshop on Cooperative and Human Aspects of Software Engineering, Gothenburg, Sweden.

Clancy, M. J., & Linn, M. C. (1999). Patterns and pedagogy. *SIGCSE Bull., 31*(1), 37-42.

Cohen, L., Morrison, K., & Manion, L. (2011). *Research Methods in Education* (Vol. 7th ed). London: Routledge.

Czerkawski, B. C., & Lyman, E. W. (2015). Exploring issues about computational thinking in higher education. *TechTrends, 59*(2), 57-65.

Dalen, M. (2011). *Intervju som forskningsmetode* (2. utg. ed.). Oslo: Universitetsforl.

de Lira Tavares, O., de Menezes, C. S., & de Nevado, R. A. (2012). *Pedagogical architectures to support the process of teaching and learning of computer programming*. Artikkel er presentert i 2012 Frontiers in Education Conference Proceedings.

Denzin, N. K., & Lincoln, Y. S. (1994). *Handbook of qualitative research*: Sage Publications.
Dillenbourg, P. (1999). *Collaborative learning: Cognitive and computational approaches. advances in learning and instruction series*: ERIC.
Dodds, Z., Libeskind-Hadas, R., & Bush, E. (2012). *Bio1 as CS1: evaluating a crossdisciplinary CS context.* Artikkel er presentert i Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education.
Dvergsdal, H. (2017, 12.12). Python - programmeringsspråk. Hentet fra https://snl.no/Python_-_programmeringsspr%C3%A5k
Ettles, A., Luxton-Reilly, A., & Denny, P. (2018). *Common logic errors made by novice programmers*. Artikkel er presentert i Proceedings of the 20th Australasian Computing Education Conference, Brisbane, Queensland, Australia.
Fernández, J. M., Zúñiga, M. E., Rosas, M. V., & Guerrero, R. A. (2018). Experiences in Learning Problem-Solving through Computational Thinking. *Journal of Computer Science & Technology, 18*.
Firebaugh, G. (2008). *Seven rules for social research*. Princeton, N.J: Princeton University Press.
Forte, A., & Guzdial, M. (2005). Motivation and nonmajors in computer science: identifying discrete audiences for introductory courses. *IEEE Trans. on Educ., 48*(2), 248-253.
Gomes, A., Carmo, L., Bigotte, E., & Mendes, A. (2006). *Mathematics and programming problem solving.* Artikkel er presentert i 3rd E-Learning Conference–Computer Science Education.
Gomes, A., & Mendes, A. (2010). *A study on student performance in first year CS courses.* Artikkel er presentert i Proceedings of the fifteenth annual conference on Innovation and technology in computer science education.
Grover, S., & Pea, R. (2013). Computational thinking in K–12: A review of the state of the field. *Educational researcher, 42*(1), 38-43.
Guzdial, M. (2010). Does contextualized computing education help? *ACM Inroads, 1*(4), 4-6.
Haberman, B., & Muller, O. (2008). *Teaching abstraction to novices: Pattern-based and ADT-based problem-solving processes.* Artikkel er presentert i 2008 38th Annual Frontiers in Education Conference.
Hagan, D., & Markham, S. (2000). Does it help to have some programming experience before beginning a computing degree program? *SIGCSE Bull., 32*(3), 25-28.
Hazzan, O., & Kramer, J. (2007). Abstraction in computer science & software engineering: A pedagogical perspective. *Frontier Journal, 4*(1), 6-14.
Hazzan, O., Lapidot, T., & Ragonis, N. (2011). *Guide to Teaching Computer Science: An Activity-Based Approach*: Springer Publishing Company, Incorporated.
Hristova, M., Misra, A., Rutter, M., & Mercuri, R. (2003). Identifying and correcting Java programming errors for introductory computer science students. *SIGCSE Bull., 35*(1), 153-156.
Hsu, T.-C., Chang, S.-C., & Hung, Y.-T. (2018). How to learn and how to teach computational thinking: Suggestions based on a review of the literature. *Computers & Education, 126*, 296-310.
Johnson, R. B., & Christensen, L. (2013). *Educational Research: Quantitative, Qualitative, and Mixed Approaches*: SAGE Publications.
Jones, B. F., Rasmussen, C. M., & Moffitt, M. C. (1997). *Real-life problem solving: A collaborative approach to interdisciplinary learning*: American Psychological Association.
Keen, A., & Mammen, K. (2015). *Program Decomposition and Complexity in CS1*. Artikkel er presentert i Proceedings of the 46th ACM Technical Symposium on Computer Science Education, Kansas City, Missouri, USA.

Krueger, R. A., & Casey, M. A. (2002). *Designing and conducting focus group interviews*: St Paul, Minnesota, USA.
Kummerfeld, S. K., & Kay, J. (2003). *The neglected battle fields of syntax errors*. Artikkel er presentert i Proceedings of the fifth Australasian conference on Computing education - Volume 20, Adelaide, Australia.
Kunnskapsdepartementet. (2018, 26.06). Fornyer innholdet i skolen. Hentet fra https://www.regjeringen.no/no/aktuelt/fornyer-innholdet-i-skolen/id2606028/
Kvale, S. (2007). *Doing Interviews*. London: England, London: SAGE Publications.
Kvale, S., & Brinkmann, S. (2015). Det kvalitative forskningsintervju (3. utg.) Oslo: Gyldendal Norsk Forlag.
LeJeune, N. (2003). Critical components for successful collaborative learning in CS1. *J. Comput. Sci. Coll., 19*(1), 275-285.
Liskov, B., & Guttag, J. (2001). *Program development in Java : abstraction, specification, and object-oriented design*. Boston: Addison-Wesley.
Lister, R., Simon, B., Thompson, E., Whalley, J. L., & Prasad, C. (2006). Not seeing the forest for the trees: novice programmers and the SOLO taxonomy. *SIGCSE Bull., 38*(3), 118-122.
Malthe-Sørenssen, A., Hjorth-Jensen, M., Langtangen, H. P., & Mørken, K. (2015). Integrasjon av beregninger i fysikkundervisningen. *Uniped*(04), 303-310.
McGill, T. J., & Volet, S. E. (1997). A Conceptual Framework for Analyzing Students' Knowledge of Programming. *Journal of Research on Computing in Education, 29*(3), 276-297.
Medeiros, R. P., Ramalho, G. L., & Falcao, T. P. (2018). A Systematic Literature Review on Teaching and Learning Introductory Programming in Higher Education. *IEEE Transactions on Education, PP*(99), 1-14.
Muller, O. (2005). *Pattern oriented instruction and the enhancement of analogical reasoning*. Artikkel er presentert i Proceedings of the first international workshop on Computing education research, Seattle, WA, USA.
Muller, O., Ginat, D., & Haberman, B. (2007). Pattern-oriented instruction and its influence on problem decomposition and solution construction. *SIGCSE Bull., 39*(3), 151-155.
Müller, L., Silveira, M. S., & Souza, C. S. d. (2018). *Do I Know What My Code is "Saying"?: A study on novice programmers' perceptions of what reused source code may mean*. Artikkel er presentert i Proceedings of the 17th Brazilian Symposium on Human Factors in Computing Systems, Belém, Brazil.
Neal, L. R. (1989). *A system for example-based programming*. Artikkel er presentert i Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.
Papert, S. (1980). *Mindstorms: children, computers, and powerful ideas*: Basic Books, Inc.
Pearce, J. L., Nakazawa, M., & Heggen, S. (2015). Improving problem decomposition ability in CS1 through explicit guided inquiry-based instruction. *J. Comput. Sci. Coll., 31*(2), 135-144.
Pevzner, P., & Shamir, R. (2009). Computing Has Changed Biology—Biology Education Must Catch Up. *Science, 325*(5940), 541-542.
Pólya, G. (1945). *How to solve it : A new aspect of mathematical method*. Princeton, N.J: Princeton University Press.
Qian, Y., & Lehman, J. (2017). Students' Misconceptions and Other Difficulties in Introductory Programming. *ACM Transactions on Computing Education, 18*(1), 1-24.
Qualls, J. A., Grant, M. M., & Sherrell, L. B. (2011). CS1 students' understanding of computational thinking concepts. *J. Comput. Sci. Coll., 26*(5), 62-71.
Raadt, M. d., Toleman, M., & Watson, R. (2004). Training strategic problem solvers. *SIGCSE Bull., 36*(2), 48-51.

Repenning, A., Webb, D., & Ioannidou, A. (2010). *Scalable game design and the development of a checklist for getting computational thinking into public schools.* Artikkel er presentert i Proceedings of the 41st ACM technical symposium on Computer science education.
Robins, A., Rountree, J., & Rountree, N. (2010). Learning and Teaching Programming: A Review and Discussion. *Computer Science Education, 13*(2), 137-172.
Robson, C. (2002). *Real world research : a resource for social scientists and practitioner-researchers* (2nd ed. ed.). Oxford: Blackwell.
Rubin, M. J. (2013). *The effectiveness of live-coding to teach introductory programming*. Artikkel er presentert i Proceeding of the 44th ACM technical symposium on Computer science education, Denver, Colorado, USA.
Sengupta, P., Kinnebrew, J., Basu, S., Biswas, G., & Clark, D. (2013). Integrating computational thinking with K-12 science education using agent-based computation: A theoretical framework. *The Official Journal of the IFIP Technical Committee on Education, 18*(2), 351-380.
Shannon, A., & Summet, V. (2015). Live coding in introductory computer science courses. *J. Comput. Sci. Coll., 31*(2), 158-164.
Shute, V. J., Sun, C., & Asbell-Clarke, J. (2017). Demystifying computational thinking. *Educational Research Review, 22*, 142-158.
Sørby, S. A., & Angell, C. (2012). Undergraduate students' challenges with computational modelling in physics. *Nordina, 8*(3), 283-296.
Universitetet i Oslo. (2018). Hentet fra https://www.uio.no/studier/emner/matnat/ibv/BIOS1100/
Urness, T. (2009). Assessment using peer evaluations, random pair assignment, and collaborative programing in CS1. *J. Comput. Small Coll., 25*(1), 87-93.
Weintrop, D., Beheshti, E., Horn, M., Orton, K., Jona, K., Trouille, L., & Wilensky, U. (2015). Defining Computational Thinking for Mathematics and Science Classrooms. *Journal of Science Education and Technology, 25*(1), 127-147.
Wilensky, U., Brady, C., & Horn, M. (2014). Fostering computational literacy in science classrooms. *Communications of the ACM, 57*(8), 24-28.
Wing, J. M. (2006). Computational thinking. *Commun. ACM, 49*(3), 33-35.
Wing, J. M. (2011). Research notebook: Computational thinking—What and why. *The Link Magazine*, 20-23.
Wood, D. F. (2003). Problem based learning. *BMJ, 326*(7384), 328-330.

Vedlegg

Vedlegg A – Intervjuguide:

Velkommen til denne økten med en gruppeoppgave, etterfulgt av en uformell samtale om programmering og modellering i biologi!

Dere har nå samtykket til dette forskningsprosjektet som handler om programmering og modellering i biologi. Prosjektet er et masterprosjekt med veiledere fra institutt for biovitenskap og fysisk institutt. Målet med prosjektet er å undersøke hvilke utfordringer studenter kan møte på når de programmerer i biologi og finne ut hvordan studentene velger å løse ulike de ulike problemstillingene. Gjennom prosjektet ønsker vi å bidra til bedre læring og motivasjon i beregningsorientert biologi.

Nå skal dere først arbeide med en oppgave der dere skal programmere i biologi. Deretter tar vi et litt uformelt intervju sammen, hvor vi snakker om hvordan dere løste oppgaven og hvilke utfordringer som dere eventuelt møtte på.

Det er frivillig å delta i denne diskusjonen, og dere kan når som helst, uten begrunnelse, trekke dere. Vi ønsker å gjøre lydopptak av både oppgave og diskusjonen. Opptaket skal bare brukes til forskningsformål og vil ikke innvirke på vurdering eller karaktersetting av dere i noen av emnene dette semesteret. Opptaket vil behandles konfidensielt; dere vil ikke identifiseres med navn eller kunne gjenkjennes på annen måte i rapporter fra forskningen. Ønsker noen å trekke seg? Hvis ikke, starter vi lydopptaket nå. Vi ønsker at dere i størst mulig grad skal **diskutere med hverandre ut fra relativt åpne spørsmål og temaer som vi tar opp.**

1. **Hva tenker dere om oppgaven dere nå arbeidet med?**
   - Hvordan var den å arbeide med?
   - Hvordan var den i forhold til hva dere pleier å arbeide med i kurset?
   - Bruker dere kunnskaper dere har fra før når dere løser oppgaver i programmering? Altså ting dere lærte på videregående, eller kan fra fritiden

2. **Hvordan gikk dere frem for å løse oppgaven?**
   - Hva er det første dere gjør når dere begynner å løse oppgaven?
   - Bruker dere noen bestemte strategier eller metoder?
   - Har dere lært noen bestemte strategier eller fremgangsmåter for å løse programmeringsoppgaver?
   - Hva tenker dere om oppgaveteksten?
   - Hvordan gikk dere frem for å finne ut hva teksten spurte om?
   - Hvordan gikk dere frem for å finne genotypen til de ulike fenotypene i oppgaven?
   - Hvilke koder valgte dere å bruke? Hvorfor det?
   - Opplevde dere noen feil i kodene? Hvordan løste dere dem?
   - Hvordan arbeider dere med programmering på egenhånd?
   - Er det noen sammenhenger mellom hvordan dere tenker når dere løste oppgave a og b, i forhold til resten av oppgaven?

3. **Var det noe dere opplevde som spesielt utfordrende i oppgaven?**
    - Hvor opplevde dere at utfordringen oppstod?
    - Hva er det som gjør dette utfordrende å arbeide med?
    - Hvordan var oppgaveteksten? Var det noen ord eller uttrykk dere lurte på?
    - Hvordan var det å finne genotypen i oppgave a?
    - Hvordan var det å regne ut sannsynligheten for arv i oppgave b?
    - Hvordan var det å finne frem riktige koder og funksjoner i oppgavene?
    - Kan dere si noe generelt om hva dere synes er vanskeligst å arbeide med når dere programmerer?
    - Hva med oppgaven vil dere si var mest utfordrende? Hva var minst utfordrende?
    - Hva tenker dere om å programmere utfra biologiske problemstillinger?
    - Føler dere at det er mer eller mindre utfordrende når dere koder ut fra en biologisk problemstilling?
    - Hva tenker dere om undervisningen i emnet?
        - Er det noe som kunne vært gjort annerledes?
        - Er det noe dere skulle ønske det var mer av i undervisningen? Hvorfor det?
        - Opplever dere at det bygger på det dere kan fra før?

4. **Hvordan påvirket denne oppgaven interesse, motivasjon og mestring?**
    - Hvordan påvirket denne oppgaven interessen for å lære mer om genotyper og fenotyper?
    - Hvordan påvirket oppgaven interessen for å jobbe mer med matematiske modellering og programmering innenfor biologi?
    - Hvordan påvirket denne oppgaven deres opplevelse av mestring i emnet BIOS 1100?
    - Er det noen deler av dette emnet som dere føler større mestring i enn andre?
    - Lærte dere noe av å arbeide med oppgaven? - Som nye begreper, koder, tenkemåter?

Vedlegg B – Samtykkeskjema

Forespørsel om deltakelse i forskningsprosjektet *Programmering og modellering i biolog: Intervju og lydopptak*

Bakgrunn Det matematisk- naturvitenskapelige fakultet ved Universitetet i Oslo la om alle sine studieprogrammer høsten 2017, og programmering og modellering er nå blitt en integrert del i hele studieløpet. Fakultetet har også et senter for fremragende utdanning, Centre for Computing in Science Education (CCSE), som støtter og forsker på innføringen av programmering og modellering i realfagsstudiene. På bachelorstudiet i biovitenskap møter studentene programmering og modellering allerede første semester i et eget emne, BIOS1100 – Innføring i beregningsmodeller for biovitenskap.

Forskningsprosjektet Formålet med prosjektet er å undersøke studenters holdninger og motivasjon for programmering og modellering i biologi, samt finne ut hvordan studentene velger å løse ulike biologiske problemstillinger ved hjelp av programmering. Gjennom prosjektet ønsker vi å bidra til bedre læring og motivasjon i beregningsorientert biologi.

Hva innebærer deltakelse i studien? Datainnsamling vil skje i form av dybdeintervju med enkeltstudenter samt lydopptak fra deler av gruppeundervisningen i BIOS1100. Intervjuene vil bli benyttet til å få en bedre forståelse av resultater fra spørreskjemaundersøkelsen (eget samtykkebrev). Lydopptak i gruppeundervisning vil bli benyttet for å kunne få en bedre forståelse for hvordan du/dere arbeider når du/dere skal løse biologiske problemstillinger med programmering og modellering.

Hva skjer med informasjonen vi samler inn? Alle data som kan være personidentifiserende vil lagres på sikre servere ved UiO. Det er kun ansvarlige for studien som vil ha tilgang til dataene. Dataene vi samler inn vil være bakgrunnsdata for flere fagartikler publisert i vitenskapelige tidsskrift og for presentasjoner på vitenskapelige konferanser. Ingen personidentifiserende data skal publiseres.

Det er tenkt å samle data fra emnet i 3 år for å kunne sammenlikne og se på endring i holdninger og motivasjon over denne perioden. Inkludert analyser og publisering er prosjektet tenkt å ha en varighet på inntil 5 år, til 2023.

Frivillig deltakelse Det er frivillig å delta i studien, og du kan når som helst trekke ditt samtykke uten å oppgi noen grunn.

Dersom du ønsker å delta i studien, signerer du skjema og lever direkte til oss. Har du spørsmål til studien, ta kontakt med Tone Fredsvik Gregers (tlf. 996 97 154). E-post t.f.gregers@ibv.uio.no.

Studien er meldt inn til Personvernombudet for forskning, NSD - Norsk senter for forskningsdata AS.

Tone Fredsvik Gregers (Førstelektor, Institutt for biovitenskap)

Samtykke til deltakelse i forskningsprosjektet «Programmering og modellering i biologi»

Jeg har mottatt informasjon om forskningsprosjektet «Programmering og modellering i biologi», og er villig til å delta i intervju og/eller lydopptak i undervisningen.

☐ Lydopptak i undervisningen

☐ Intervju

_____

Signatur

Vedlegg C – Oppgave som ble bruk under datainnsamling

Exercise 6: Autosomal Recessive Inheritance

Autosomal recessive disorder appears in individuals with two abnormal copies of a gene. The genes are located on an autosome, which is a chromosome that is not a sex chromosome. Humans usually have 22 pairs of autosomes and one pair of sex chromosomes. The inherited trait of recessive disorders usually skip a few generations, while dominant traits are usually present in every generation. Therefore, the individual that inherits an autosomal recessive disorder must inherit the abnormal alleles from parents that are carriers. Examples of this disorder are cystic fibrosis, sickle cell disease, and phenylketonuria.

This exercise is about a three generation family where the grandchildren are affected by an autosomal recessive disorder. We are going to investigate their genetics and probabilities of passing on the condition.

a) The parents and grandparents show no signs of the somatic recessive condition. Given this statement and the above information, can you identify the genotypes of the parents and the grandparents?

b) What is the probability of having three affected grandchildren? Print the answer to screen.

c) Now that you are familiar with the probabilities of inheriting the disease allele, you are going to check the genotypes and phenotypes of 5 grandchildren. You only need to consider the parental alleles and their offspring.

Write a program that iterates over the 5 children and randomly generates the child's genotype from the parental alleles. Remember, if the child has the dominant allele present it will possess a normal phenotype. Print the genotypes and corresponding phenotypes for each child to screen.

Vedlegg D – Løsningsforslag på oppgave som ble brukt under datainnsamling

Oppgave a

Foreldrene må være bærere og har heterozygot genotype, Aa.

For besteforeldrene må minst en av dem være bærer til hver av foreldrene. Enten kan begge være heterozygot bærer, med Aa, eller så er en heterozygot, og en homozygot dominant, AA.

Oppgave b

Siden ingen av foreldrene eller besteforeldrene viser tegn på sykdommen, så blir sannsynligheten for at foreldrene blir syke ekskludert fra regnestykke.

Det er 2/3 sjanse for at foreldrene blir heterozygote bærere, og 1/3 sjanse for at de blir homozygot dominante. For å få ett sykt barn, må begge foreldre være bærere. Sannsynligheten for at foreldrene får et sykt barn er 1/4.

Sannsynligheten for ett syk barnebarn er = 2/3 *2/3*1/4 = 1/9.
Sannsynligheten for tre syke barnebarn er = 1/9 * 1/9 * 1/9 = 1/729

Oppgave c

```
from pylab import *

parent_1 = ["A", "a"] # carrier
parent_2 = ["A", "a"] # carrier

genotype_children = []
phenotype_children = []

children = 5
child = 0

while child < children:
    allele_1 = choice(parent_1)
    allele_2 = choice(parent_2)

    genotype = [allele_1, allele_2]

    if "A" in genotype:
        phenotype = "normal"
    else:
        phenotype = "affected"

    genotype_children.append(genotype)
    phenotype_children.append(phenotype)
    child += 1

print(genotype_children)
print(phenotype_children)
```