Programmeringsoppgaver og auto-scoring: Inspera vs. andre løsninger

Guttorm Sindre[1][0000-0001-5739-8265]

[1] Institutt for datateknologi og informatikk (IDI), NTNU
guttorm.sindre@ntnu.no

Sammendrag. Retting av store antall eksamensbesvarelser i programmeringsfag innen stramme sensurfrister er krevende. Selv om oppgavetypen «Programmering» i Inspera gir klare fordeler framfor håndskrevet kode, er den manuelt rettet. I 2021 lanserte Inspera en ny oppgavetype, «Kompilering», hvor poeng kunne gis automatisk basert på testsuiter. Dessverre var oppgavetypen beheftet med en del problemer, og den er nå trukket tilbake for videreutvikling, dermed ikke lenger tilgjengelig for bruk i eksamen. Denne artikkelen beskriver våre erfaringer med oppgavetypen «Kompilering» og diskuterer hvilke behov for forbedringer som ville være ønskelig for at bruk av denne skulle kunne bli vellykket - særlig for eksamen, men også potensielt for bruk under formativ vurdering underveis i semesteret. Det gjøres en sammenligning med andre systemers implementasjon av lignende oppgavetyper, f.eks. Moodle CodeRunner, basert på praktisk utprøving og litteraturstudium av erfaringer med bruk i andre land. Artikkelen diskuterer også i hvilken grad karaktersetting kan basere seg fullt ut på automatisk poenggivning av kode, eller hvorvidt man fortsatt vil trenge manuell gjennomgang, for eksempel for å kunne gi noe poeng til studenter som har kode som ikke fungerer, men hvor løsningen likevel inneholder mye riktig tankegang.

Nøkkelord: programmering, autoretting, digital eksamen, formativ vurdering.

1 Introduksjoon

Auto-retting eller auto-scoring av studenters programkode kan være interessant av flere grunner. Med tanke på eksamen og andre summative vurderinger kan det gi mer objektiv karaktersetting, samt gjøre det lettere å fullføre sensur innen de stramme fristene som den norske Universitetsloven pålegger, særlig i emner med mange studenter. Med tanke på formativ vurdering kan auto-retting gi muligheter for øyeblikkelig tilbakemelding til studenter om hvorvidt kode de har skrevet på øvinger eller selvtester har fungert eller ikke, muligens også med henvisning til materiale de kan se på hvis feil svar tyder på at de har misforstått sentrale konsepter eller problemløsningsmønstre.

Det fins en rekke oppgavetyper som trivielt kan autorettes. Av sjangre som typisk er brukt til eksamen i programmering [1] kan programforståelse (f.eks. spørsmål av typen: Hva blir utskriften fra dette programmet?) lett gis som flervalg, eller som fyll inn tall og fyll inn tekst som også er greit å auto-rette. Likeledes fins det oppgavetyper

med komplettering av kode. Oppgaver hvor kodelinjer er gitt i omstokket rekkefølge og skal plasseres riktig av studenten, kalles gjerne *Parson's problems* innen programmering, jfr. [2, 3]. Eksperimenter har indikert at slike oppgaver både kan ha god læringseffekt og god korrelasjon med evne til å skrive kode [4, 5]. Oppgaver hvor noe kode er gitt men hull skal fylles, har også lange tradisjoner og empirisk dokumenterte pedagogiske meritter [6]. Studenters prestasjoner på slike oppgaver har vist seg å korrelere godt med prestasjoner på oppgaver med skriving av kode [7-9], og Cheng & Harrington [7] antyder endatil at man kan basere karaktersetting på slike oppgaver heller enn skriving av kode hvis overkommelig tidsbruk til sensur er viktig.

Samtidig har programkode langt bedre forutsetninger for å kunne rettes automatisk enn langsvar i naturlig språk. Siden programmeringsspråk har en presis, formell syntaks, kan kode analyseres automatisk på vesentlig enklere måte enn naturlig språk. Hvis koden kan kjøres, vil man kunne observere hvorvidt den returnerer korrekt svar i forhold til test cases, samt også eventuelt kunne måle kjøretid og annen ressursbruk. Selv for kode som ikke kjører korrekt, kan statisk analyse si noe om kodestruktur, kodestil, syntaks, osv.

Ikke overraskende fins det da også en del e-læringsverktøy som støtter automatisk retting av kode. Dels er dette tilleggsmoduler til generelt tilgjengelige verktøy, slik som CodeRunner [10] for Moodle og nbgrader [11] for Jupyter Notebook, dels kan det være verktøy som er brukt bare mer lokalisert i enkelte fagmiljøer. Når det gjelder verktøy for digital eksamen brukt i Norge, introduserte Inspera i 2021 en oppgavetype kalt «Kompilering» hvor kode kunne kjøres mot testsuiter og poeng dermed kunne gis ut fra hvor mange tester den passerte. Denne oppgavetypen ble brukt på eksamen i IT Grunnkurs (Python) ved NTNU høsten 2021 – men fungerte ikke helt etter forutsetningene. På grunn av problemer med oppgavetypen (muligens problemer observert under vår eksamen, muligens andre) er den deretter blitt fjernet fra produksjonsversjonen av Inspera og puttet tilbake til utvikling, slik at den forhåpentligvis vil være betydelig forbedret når den kommer tilbake.

Selv om den aktuelle funksjonaliteten altså for øyeblikket ikke er tilgjengelig, er det likevel interessant å analysere og dele våre erfaringer med denne oppgavetypen. Forskningsspørsmålene våre er dermed som følger:

FS1) Hvordan fungerte «Kompilering» i Inspera sammenlignet med andre lignende oppgavetyper, som f.eks. Moodle CodeRunner?

FS2) Hvordan var studenters prestasjoner med oppgaver av denne typen? Særlig interessant: I hvilken grad fordelte prestasjoner seg over hele spektret, heller enn binært mellom studenter som hadde riktig kode (og dermed passerte alle tester, full score) og feil kode (og dermed ikke passerte noen tester, null score)?

FS3) Hva slags forbedringer kan man ønske i oppgavetypen «Kompilering» - eller lignende oppgavetyper fra andre leverandører av produkter innen LMS og digital eksamen?

Resten av artikkelen er strukturert som følger: Seksjon 2 beskriver noen andre systemer for auto-retting av kode og diskuterer vitenskapelige publikasjoner om erfaringer med bruk av disse. Seksjon 3 beskriver deretter våre erfaringer med bruk av oppgavetypen «Kompilering» i Inspera. Seksjon 4 presenterer resultatene fra en analyse av

studentenes prestasjoner med slike oppgaver, hvorpå seksjon 5 gir en diskusjon og konklusjon.

2 Litteraturgjennomgang, andre systemer

2.1 Moodle CodeRunner

Moodle er et open source LMS med betydelig markedsandel (dog lavere enn Canvas), og er særlig mye brukt innen teknologi- og realfagsutdanning [12]. CodeRunner [10] er en tilleggsmodul til Moodle utviklet ved University of Canterbury, New Zealand. Modulen gjør det mulig for faglærer å lage programmeringsspørsmål som studenten kan besvare enten ved å skrive kode direkte i Moodle eller lime inn kode skrevet fra en editor. I Moodle kan koden sjekkes mot testsuiter, både av studenten under arbeid med prøven, og av fagstaben i forbindelse med automatisk scoring av levert kode.

Figur 1 viser eksempler på skjermbilder fra CodeRunner, med en svært enkel oppgave (funksjon for å kvadrere tallet som gis inn). Øvre venstre viser en korrekt løsning, og nedre venstre den tilbakemeldingen studenten vil få ved å trykke «Check». Øvre høyre viser en feilaktig løsning som kun virker for de oppgitte testtilfellene. Som man kan se av bildet nedre høyre, feiler denne på grunn av at det fins flere testtilfeller enn de oppgitte, hvorav noen vises i tilbakemeldingen (-4 og 0) mens andre forblir skjult. De skjulte testtilfellene forhindrer altså at studenter kan lure systemet med løsninger à la øvre høyre (som kanskje ikke er særlig aktuelt her hvor korrekt løsning er enda enklere, men ville være en aktuell trussel for vanskeligere oppgaver.

Faglærer har stor fleksibilitet i å sette opp ulike regler for automatisk poengscore på levert kode. Eksemplet i skjermbildet (nedre høyre) indikerer at alle testtilfeller må passeres for at man skal score poeng, men man kan alternativt velge å gi partiell score hvis noen testtilfeller er passert. Man kan også sette opp at poengscore gradvis reduseres for hver bruk av «Check»-tasten med feilaktig kode, f.eks. 10% reduksjon hver gang. Faglærer har stor fleksibilitet i å lage oppgavetyper og regler for poenggivning utover standard oppførsel – vel å merke hvis faglærer selv er i stand til å skrive Python-kode og JSON for hvordan CodeRunner skal oppføre seg.

CodeRunner har etter hvert blitt brukt ved mange universiteter, og noen faglærere har også publisert om sine erfaringer med verktøyet. Croft og England [13] beskriver sine erfaringer med bruk av CodeRunner til formative vurderinger underveis i semesteret i to programmeringskurs for førsteårsstudenter (ett i Python, ett i C++) ved Univ. Coventry. Erfaringene var særlig positive i Python-kurset (økt tilfredshet, bedre eksamensprestasjoner), noe mindre i C++-kurset, hvor forfatterne mener at de i for liten grad klarte å gi partiell score på delvis riktig kode. Bralic et al. [14] beskriver sine erfaringer fra Univ. Applied Sciences, Velica Gorica, Kroatia, med overgang fra manuelt rettede programmeringsoppgaver på eksamen til bruk av CodeRunner med automatisk retting, i store trekk en suksess både med økt tilfredshet hos studenter og spart tid for faglærere. Lignende erfaringer kommer fram hos Pringuet et al. [15] fra det amerikanske universitetet i Bahrain, som brukte CodeRunner både til formative vurderinger og eksamen. Bortsett fra noe misnøye med standardfunksjonaliteten (som tilsa at de neste år kanskje ville prøve å programmere noen tilpasninger av verktøyet selv)

observerte man både økt tilfredshet blant studentene relatert til hurtig automatisk tilbakemelding på programmeringsbesvarelser, og økt tilfredshet blant faglærere, både ved å kunne prøve ut nye oppgavetyper og ved å spare tid på rettearbeid, slik at de til gjengjeld hadde mer tid til overs for å følge opp studenter som slet med å lære seg programmering.

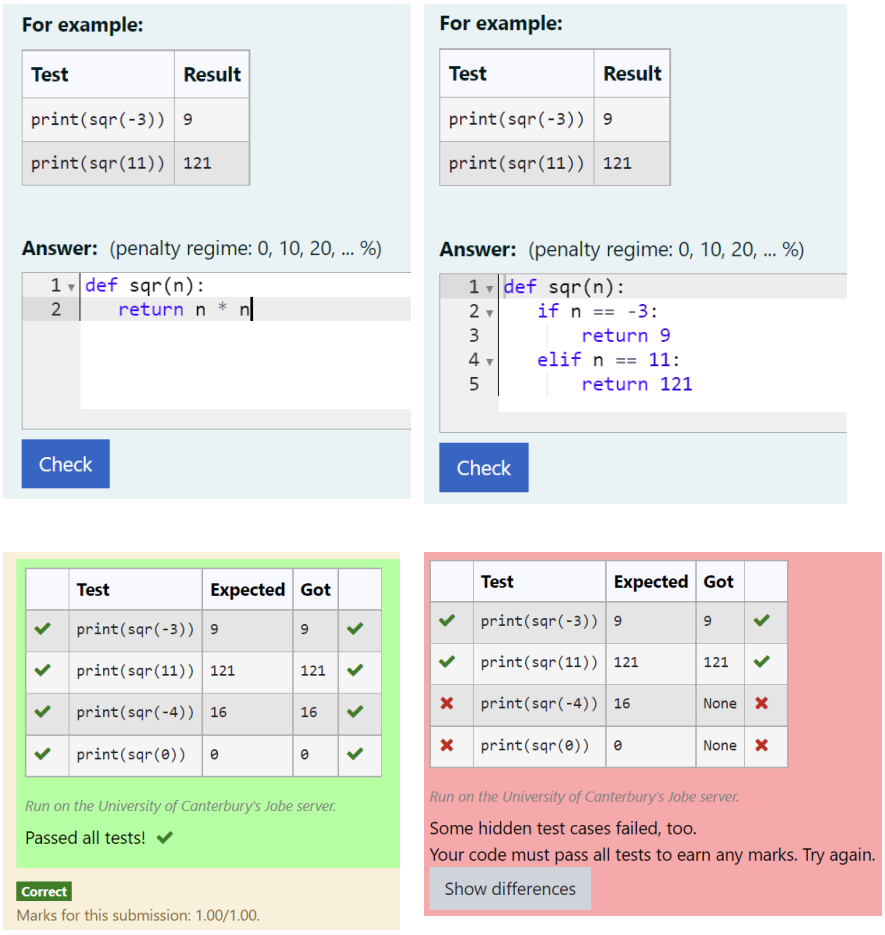


Fig. 1. Eksempel på skjermbilder fra CodeRunner, https://coderunner.org.nz/mod/quiz/.

2.2 Andre system, særlig nbgrader

Jupyter Notebooks er mye brukt i realfagsundervisning på universitetsnivå, og nbgrader [16] er en tilleggsmodul til denne som støtter manuell og automatisk poenggivning og karaktersetting både for programmeringsoppgaver og annet innhold. For programmeringsoppgaver kan man omtrent som for Moodle CodeRunner kjøre koden mot testtilfeller, og særlig oppgaver som handler om å skrive en funksjon med returverdi vil egne

seg godt for dette. Mens Moodle CodeRunner både fokuserer på tilbakemelding til studenten underveis og poenggivning etter levering, er nbgrader dog primært fokusert på det siste, men siden det er åpen kildekode vil det på samme måte som for Moodle CodeRunner være stor fleksibilitet for tilpasninger for faglærere som selv er i stand til å programmere.

Selv om Jupyter og nbgrader er mye brukt, er det ikke publisert veldig mye forskning på erfaringene med det. Zhao og Xia [17] diskuterer sine erfaringer med bruk av Jupyter i undervisning, inkludert positive erfaringer med nbgrader i form av redusert tidspress på faglærere med tanke på retting av besvarelser. Manzoor et al. [18] presenterer dog et opplegg ved Virginia Tech hvor det valgte å bruke et annet autoscoringsverktøy enn nbgrader for å gi poeng til kode studentene hadde levert i Jupyter. De valgte i stedet å tilpasse og bruke Web-CAT fordi dette ga enklere interoperabilitet med LMS (Canvas) da de ønsket å få overført poengsummer og tilbakemeldinger dit på en enkel måte. Artikkelen deres nevner også flere andre systemer for auto-scoring av programkode.

3 Erfaringer med Inspera "Kompilering"

3.1 Funksjonalitet

Avveiningene i artikkelen til Manzoor et al. nevnt mot slutten av forrige seksjon viser at det er mange ulike hensyn som kan spille inn i valget av verktøy, blant annet at det skal passe sammen med andre verktøy man har. Dette vil typisk være situasjonen i Norge også, man har allerede et LMS (typisk Canvas eller Blackboard) og et digitalt eksamenssystem (Inspera eller WISEflow), og særlig for eksamen kan bruk av tilleggsverktøy – f.eks. en programmeringsomgivelse – være problematisk med tanke på sikkerhet mot fusk. Å bruke Moodle CodeRunner eller Jupyter Notebook med nbgrader i forbindelse med en eksamen ville derfor ikke være trivielt, og det var store forhåpninger knyttet til at Inspera skulle legge til en oppgavetype som kunne gi lignende muligheter, både for at studenter kunne kjøre kode mot testsuiter underveis i eksamen, og at testtilfeller kunne brukes til automatisk poenggivning. På den positive siden hadde oppgavetypen "Kompilering" både mulighet for å ha åpne testtilfeller (synlige for student, for testing underveis i eksamen) og skjulte testtilfeller (for sensor), slik at man ikke skulle kunne «lure» en hvilken som helst oppgave bare med en if-elif-else som akkurat tok for seg testtilfellene, samt også mulighet for oppgaveforfatter til å stille inn hvorvidt det kun skulle gis poeng hvis alle testtilfeller var passert, eller om studenten kunne få delvis score for hvert passerte testtilfelle. Om det ikke ville gi studenten en like autentisk opplevelse som å kode i en ordentlig programmeringsomgivelse, ville iallfall muligheten til å teste koden underveis gi en mer autentisk opplevelse enn den gamle oppgavetypen «Programmering», hvor det eneste man fikk var syntaksbasert fargelegging av koden.

Imidlertid hadde også oppgavetypen en god funksjonelle mangler, for eksempel sammenlignet med Moodle CodeRunner, og mange av disse fremsto tydelig for oss allerede før eksamen. Noen av de viktigste svakhetene presenteres under:

Testkode, synlighet og editerbarhet. Testtilfellene i Inspera var utelukkende basert på input( ) og print( ), og siden verktøyet ikke er åpen kildekode, var det heller ikke mulig for faglærer å tilpasse det til noe annet. Hvis programmeringsoppgaven man tenkte å gi, ikke i seg selv naturlig inkluderte input( ) og print( ), måtte faglærer legge til dette som forhåndsdefinert kode i oppgaven for at tester skulle kunne kjøres. Figur 2 illustrerer dette, med en hypotetisk, liten programmeringsoppgave hvor studentene skal skrive en funksjon som får inn en liste med heltall og returnerer antall oddetall i denne listen. Venstre side viser en rett fram løsning på denne, mens høyre side viser løsningen tillagt en input() i første linje og en print() i siste linje for å kunne kjøre koden mot testtilfeller i Inspera. Merk at input() er uten noe prompt, og studenten vil ikke bli bedt om å skrive inn verdier for denne inputen under testing av koden – i stedet vil en slik input i starten av koden konsumere inn-verdier som er gitt for hvert enkelt testtilfelle. Her vil dette være ulike lister som kan være gitt som input, og for å få dette til å bli en liste som kan brukes videre i koden heller enn bare en tekststreng, må denne settes inni en eval(). Selv om det er krøkkete sammenlignet med å kunne basere testtilfellene direkte på argument-returverdi-par for funksjonen, kunne dette ha fungert greit nok siden faglærer lett kan legge inn første og siste setning som forhåndsdefinert kode. Imidlertid er det ytterligere problemer med Inspera sitt valg: *forhåndsdefinert kode er alltid synlig og mulig å editere for studentene*. At kode er synlig og kan endres, ville ha vært nyttig hvis oppgaven inneholdt feilaktig kode som det var opp til studenten å korrigere, men ikke når den forhåndsdefinerte koden kun er nødvendig for testkjøring. Studentene har ingen nytte av å se disse kodelinjene, som i beste fall kan være forvirrende (f.eks. siden eval() ikke er pensum), og i verste fall kan studentene være uheldige å endre noe i den forhåndsdefinerte koden, slik at testene ikke lenger virker. En student kan da ha skrevet selve funksjonen riktig, men likevel komme ut med null poeng. Det som ville ha vært en klart bedre løsning her, ville være at faglærer / oppgaveforfatter for hvert tilfelle av forhåndsdefinert kode kunne velge: skal koden være synlig for studentene eller ikke, og hvis synlig, skal den være mulig å endre eller ikke?

```
def numOdd(intList):
    count = 0
    for num in intList:
        if num % 2:
            count += 1
    return count
```

Mulig korrekt løsning fra student

```
intList = eval(input())

def numOdd(intList):
    count = 0
    for num in intList:
        if num % 2:
            count += 1
    return count

print(numOdd(intList))
```

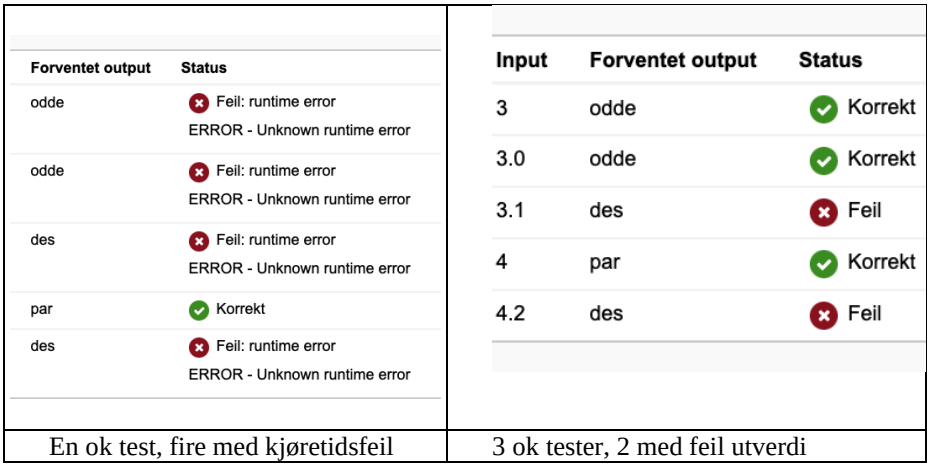
Løsning pluss testkode først og sist

Fig. 2. Løsning på kodeoppgave (v.s.), og inkludert input() og print() for testing (h.s.).

Tilbakemeldinger ved feil. Disse var svært lite informative i Inspera. I Moodle CodeRunner får man ved kjøretidsfeil opp melding om hva slags feil (f.eks. NameError) og hvilken kodelinje den inntraff i. Ved kode som gir feil resultat, får man opp en tabell

over hvilke verdier som var forventet fra testen, versus hvilke man selv hadde (kolonnene «Expected» og «Got» nederst til høyre i Figur 1. For «Kompilering» i Inspera var feilmeldingene mye vanskeligere å bli klok på. For testtilfeller med kjøretidsfeil får man kun et rødt kryss og meldingen «Unknown runtime error», uten noe hint om kodelinje, jfr. venstre side i Figur 3. For tester hvor utverdi var feil, fins det ingen kolonne som forteller hvilken verdi studenten selv fikk (à la «Got»-kolonnen i CodeRunner), i stedet får man bare vite hvilken verdi man skulle hatt – og at man hadde «Feil». Denne vagheten i tilbakemeldingene gjør det mye vanskeligere for studenter å skjønne hva som kan være galt med koden, og dermed vanskeligere å rette opp feil, enn om man hadde gjort tilsvarende test i Moodle CodeRunner eller i en skikkelig programmeringsomgivelse.

Man kan selvsagt tenke seg tilfeller hvor faglærer *ikke* ønsker at studentene skal få informative feilmeldinger fra systemet, og heller ikke ønsker at de skal kunne se hva slags verdier de selv fikk side om side med forventet output. Men i det minste burde dette da ha vært en innstilling hvor oppgaveforfatter kunne bestemme hvor mye informasjon studenten skulle få ut om kode som ikke fungerte som forventet, f.eks. om en kolonne for egen output skulle vises eller ikke. Vår bruk av oppgavetypen var på en hjemmeeksamen, så studentene hadde mulighet til å kjøre koden i en mer egnet editor for å se mer presist hva som var feil, men dette ville gi noe ekstra tidsbruk for dem fordi de da også ville måtte legge inn kode for testsuitene. Hvis det i stedet var en eksamen under tilsyn, hvor tredjepartsverktøy ikke kunne brukes, vil de vage tilbakemeldingene gjøre at det blir vanskelig for studentene å korrigere koden sin og testen blir dermed mindre autentisk.



En ok test, fire med kjøretidsfeil | 3 ok tester, 2 med feil utverdi

Fig. 3. Feilmeldinger på kjøretidsfeil og feil utverdi i Inspera «Kompilering»

3.2 Robusthet og ytelse

I tillegg til noen svakheter i funksjonalitet som nevnt over, var det også problemer med ytelse og robusthet. Når studenten trykker for å teste koden, sendes den til Insperas server og testes der (sannsynligvis i en dertil egnet sandkasse). Dette tok imidlertid litt tid, typisk minst 15 sekund fra man klikket til man fikk resultatet, også for svært kort kode. Dette vil dra ned arbeidstempoet for studenter som sliter med å finne og korrigere en feil og dermed ender opp med å teste samme kode mange ganger.

Langt verre var det likevel at testingen brøt fullstendig sammen under deler av eksamen når mange studenter samtidig jobbet med oppgaver av typen «Kompilering». Igjen ble dette problemet ikke så ille som det kunne ha vært i og med at det var en hjemmeeksamen og studentene dermed kunne kjøre koden sin i programmeringsomgivelser som IDLE, Thonny eller VSCode, eller bruke Jupyter Notebook eller andre foretrukne verktøy og bare til slutt lime koden sin inn i svarvinduet i Inspera uten å teste den der. På en skoleeksamen uten mulighet til å bruke tredjepartsverktøy, ville fravær av testmulighet ha gjort at oppgaver av typen «Kompilering» egentlig ville ende opp med bare å være det samme som den gamle typen «Programmering» fra studentenes synsvinkel. Sensor ville fortsatt kunne bruke testsuitene etterpå for automatisk poenggivning, men hvis studentene ikke hadde reell mulighet til å teste koden under eksamen, ville svært mange ha kode som ikke kjørte, slik at man likevel i stor grad ville måtte gjennomgå manuelt oppgaver som egentlig skulle rettes automatisk.

3.3 Eksamensprestasjoner

Det var 8 deloppgaver på eksamen i IT Grunnkurs høsten 2021 som var av typen «Kompilering», nemlig 2 a, b, c, g, h, i og 3 a, b. Til sammen utgjorde disse 33% av eksamen med tanke på vekting. Det aktuelle semesteret ble IT Grunnkurs gitt som to ulike emnekoder, TDT4109 som tas av studenter på linjene data, informatikk og andre nærliggende linjer (ca. 500 studenter) og TDT4110 som ble tatt av alle andre linjer (over 2000 studenter). Eksamen var felles for begge emnekodene og ble avholdt som hjemmeeksamen med bruk av Inspera, men uten bruk av Safe Exam Browser, dvs. studentene kunne bruke internett, programmeringsverktøy etc. samtidig som de løste eksamen (bruk av lockdown browser på hjemmeeksamen har uansett liten hensikt, siden kandidatene lett kan aksessere internett med annet utstyr enn akkurat den PCen de kjører Inspera på). På første side av eksamen sto det en del regler relatert til fusk som studentene måtte bekrefte at de hadde lest og akseptert. Dessuten sto det et spørsmål om man samtykket i at resultatene kunne brukes til forskning, hvor det var fritt valg om man ville svare positivt eller negativt. Resultatene i tabellen baserer seg kun på de besvarelsene hvor studenten samtykket til bruk i forskning, som man kan se av tabellen, var dette 435 studenter (88%) for TDT4109 og 1823 studenter (87%) for TDT4110.

Hver deloppgave har en kolonne i tabellen. De to første deloppgavene (2a, 2b) var svært enkle, med score over 90% for TDT4109 og over 80% for TDT4110. 2c, 3a, 3b var noe vanskeligere, med snittscore rundt 85 for 4109 og rundt 70 for 4110. Aller vanskeligst var 2g, fulgt av 2h og i, merket med rødt og oransje henholdsvis. På alle deloppgaver lå snittscore for emnekoden TDT4110 en del lavere enn for TDT4109. Dette er ikke overraskende siden TDT4109 blir tatt av studieprogram innen data,

informatikk og nærliggende retninger, mens TDT4110 hadde et mye større spenn av studieprogrammer, sannsynligvis med mer varierende motivasjon for å lære programmering, samt mer variasjon i opptakspoeng som man trengte for å komme inn på studiene.

Det man kunne frykte ved oppgaver av typen «Kompilering» - jamfør forskningsspørsmål 2 – er at resultatet på hver deloppgave ville bli svært bimodalt, at studenter enten hadde kode som virket og fikk full score, eller at den ikke virket med null score som resultat. Tabell 1 indikerer imidlertid at det også var en betydelig andel studenter som hadde delvis score på oppgavene, dvs. koden deres passerte noen tester, men ikke alle. Dette er iallfall tilfelle for oppgavene 2c, 2g, 2h, 2i og 3a (jfr. prosenttallene som er gitt med fete typer i tabellen) – og særlig for den aller vanskeligste oppgaven 2g, hvor prosentandelen med delvis score er over 50 for begge emnekoder. Det er derfor bare oppgave 2a, 2b og 3a hvor svært få har delvis score. De to første av disse var svært enkle oppgaver hvor en meget høy andel av studentene hadde alt rett, som gjør det naturlig at få hadde delvis score. Kun for den siste (3b) er det slik at andelen som hadde delvis score er vesentlig lavere enn de som hadde null score.

Tabell 1. Studenters prestasjoner på oppgaver av typen "Kompilering", høst 2021

Emnekode	#stud (ja%)	Deloppgaver av typen "Kompilering" (33%)								
TDT4109	435 (88%)	2a	2b	2c	2g	2h	2i	3a	3b	Total
Snittscore, %		94	97	85	47	67	71	85	83	76
% stud alle tester ok		93	97	77	24	57	65	70	81	11
% stud delvis score		2	1	**14**	**54**	**24**	**15**	**22**	3	87
% stud alle tester feil		5	2	9	22	19	20	8	16	1
TDT4110	1823 (87%)	2a	2b	2c	2g	2h	2i	3a	3b	Total
Snitt		84	85	71	36	46	50	70	67	61
Rett		81	84	60	12	37	43	52	65	5
Delvis		6	1	**21**	**58**	**21**	**15**	**30**	2	93
Feil		13	15	19	30	42	42	18	33	2

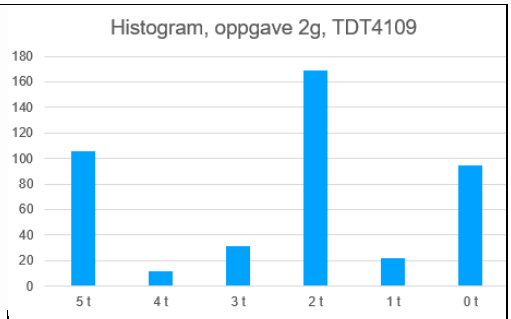
Det kan derfor være interessant å se spesielt på noen av deloppgavene, særlig den vanskeligste 2g (hva var det ved denne som til tross for relativt høy vanskegrad gjorde

at mange studenter likevel oppnådde en delvis score?), og noen av oppgavene i midtskiktet (Hva kan være mulig årsak til at 3b hadde langt færre med delvis score enn 2c og 3a som hadde omtrent samme vanskegrad?)

Oppgave 2g handlet om å skrive en funksjon som får inn ei liste med tall, og som gjør muterende endring i samme listeobjekt ved å fjerne påfølgende duplikater (men **ikke** alle duplikater). For eksempel, lista [1, 1, 2, 2, 2, 1, 3, 3] skal bli endret til [1, 2, 1, 3] – dvs., fortsatt to enere i lista, fordi de ikke er påfølgende. Dette kan løses på flere måter, hvor figur 4 viser to ganske rett fram muligheter, en med for-løkke, hvor det er mest hensiktsmessig å traversere lista bakfra for å unngå at det blir tull med indeksering når man fjerner elementer, og en hvor man starter fremst i listen men bruker while-løkke, og hvor indeks kun økes i tilfeller hvor man ikke fjerner elementet (siden man ved fjerning får aksess til det neste elementet på den indeksen man allerede står). Andre løsninger var også mulig, mange prøvde for eksempel å lage ei ny liste hvor man bare puttet inn elementer hvis de ikke hadde samme verdi som forrige element, for til slutt å mutere denne nye lista inn i det gamle listeobjektet (hvis vellykket; eller unnlot å gjøre noen mutering, som ville gi en feilaktig løsning). Av histogrammet til høyre i figur 4 kan man se at selv om det var mange som oppnådde delvis score, så klumpet resultatene seg i all hovedsak sammen på tre mulige utfall: alle tester ok (venstre), 2/5 tester ok (høyeste kolonne i midten) eller ingen tester passert (høyre) – bare et fåtall kandidater hadde besvarelser som passerte 4, 3 eller 1 test. Dette vil åpenbart ha sammenheng med utvalget av testtilfeller for oppgaven.

```
def remove_consec_duplicates(L):
    for i in range(len(L)-1, 0, -1):
        if L[i] == L[i-1]:
            del L[i]

def remove_consec_duplicates(L):
    i = 1
    while i < len(L):
        if L[i] == L[i-1]:
            del L[i]
        else:
            i += 1
```



Mulige rett-fram løsninger	Histogram, full score til venstre

Fig. 4. Oppgave 2g, eksempelløsning og histogram for prestasjoner

Oppgave 2g hadde følgende skjulte testtilfeller som ble brukt til poengscore (i form av lister inn til funksjonen): [1], [1,3,5], [2,3,2,2,4], [1, 1, 1, 1, 1], [2, 4, 4, 4, 2, 2, 4, 4, 2]. Av disse skulle de to første passere uendret gjennom funksjonen, mens de tre siste skulle bli endret til henholdsvis [2, 3, 2, 4], [1], [2, 4, 2, 4, 2]. Grunnen til at mange kandidater har greid akkurat to tester her, er dermed at mange har fått til en funksjon som virker for lister som ikke skal endres, men ikke fått til endringen – eller klart å endre bare en lokal listevariabel i funksjonen men ikke mutert selve inn-lista.

Andre oppgaver viste lignende tendenser: Der hvor det var en del studenter som hadde fått delvis poengscore, var det ofte visse antall korrekte tester som var klart mer dominerende enn andre. Overordnet fordelte prestasjonene seg likevel jevnere utover, fordi det i noen grad varierte hvilke studenter som klarte alle testene på de ulike oppgavene. Figur 5 under viser hvordan «karakterer» fordelte seg hvis man ser på kun oppgavetypen «Kompilering» på den aktuelle eksamen. «Karakterer» står i hermetegn her fordi eksamen hadde bestått / ikke bestått, og selv om det hadde vært bokstavkarakterer, ville ikke en enkelt oppgavetype som utgjorde bare 1/3 av eksamen ha fått separat karakter. Det kan likevel være interessant å se på for et visst overblikk over prestasjonen. Emnekoden TDT4109 (for studieprogram som skulle antas å være spesielt motivert for programmering) hadde en topp på «A», mens TDT4110 med mer varierte studieprogram hadde en topp på «B». Selv for den «svakere» emnekoden er ikke resultatene dårlige.

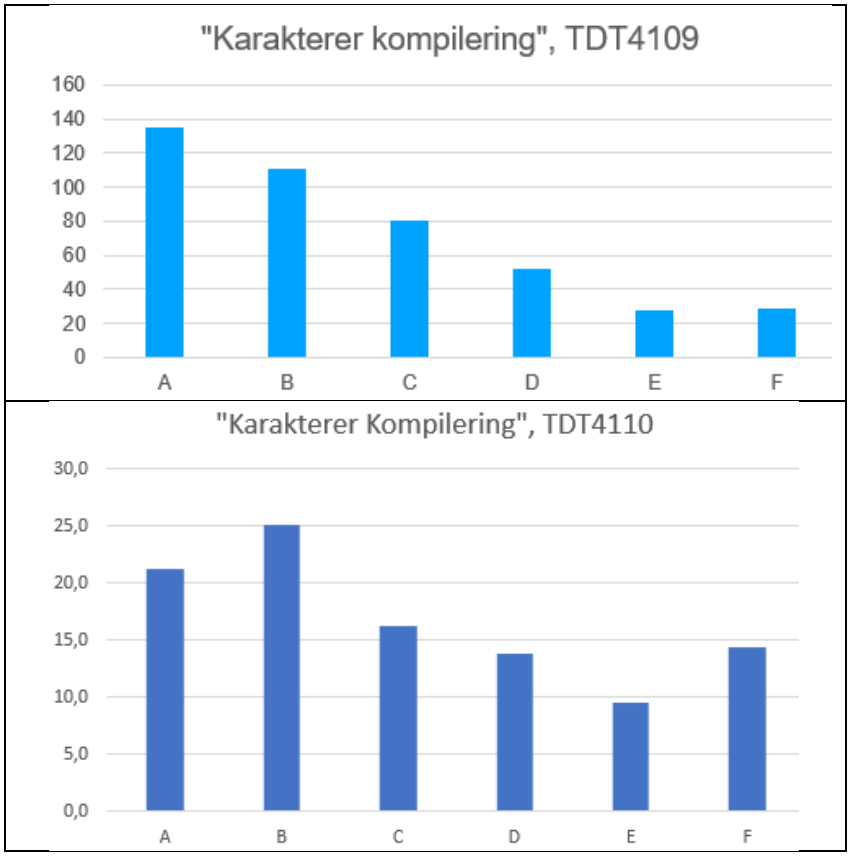


Fig. 5. «Karakterfordeling» for oppgavene av type «Kompilering»

4 Diskusjon og konklusjon

Basert på histogrammene i figur 5 ser resultatene på ingen måte stygge ut for oppgavetypen «Kompilering», på tross av tekniske problemer underveis i eksamen ser det ut til å ha gått greit for de fleste studentene. Det som reddet situasjonen i forhold til de tekniske problemene, var nok at det var hjemmeeksamen, slik at studentene kunne skrive kode i sin foretrukne programmeringsomgivelse – og der teste om den virket eller forutsetningene – og så bare lime den inn i Inspera etterpå og la være å bruke testmuligheten der (som likevel sviktet på grunn av ytelsesproblemer). Hvis det hadde vært en eksamen hvor Inspera var eneste testmulighet, og denne testmuligheten ikke virket, ville nok resultatet ha blitt vesentlig dårligere – da langt flere studenter ville hatt kode som ikke fungerte, litt i tråd med erfaringene fra [19] hvor manglende mulighet til testing av kode underveis gjorde at få studenter hadde kode som virket.

Noen av utfordringene som har vært diskutert i denne artikkelen gjelder svakheter i Inspera sitt design av oppgavetypen «Kompilering». Det er imidlertid viktig å merke seg at selv om Inspera skulle komme tilbake med en vesentlig forbedret versjon av oppgavetypen, for eksempel med bedre tilbakemelding til studenten underveis, bedre opplegg for poenggivning, og tilstrekkelig robusthet og ytelse for å takle mange samtidige prøvetakere, vil kvaliteten av en vurdering likevel stå og falle med god oppgavedesign fra fagstaben. Som histogrammet for oppgave 2g indikerer, var ikke dette helt vellykket. Ved manuell retting vil poengene typisk spre seg mer utover, f.eks. at ganske mange ville få 4 av 5 poeng for en løsning som nesten var riktig, men med noen småfeil, eller 3 av 5 poeng for en ganske riktig løsning, men med litt større feil. Selv om mange har fått 2 av 5 poeng, betyr dette heller ikke at alle disse besvarelsene holder samme kvalitet. Noen kan ha vært veldig nærme en løsning på de tre testene de manglet, mens andre kan ha hatt en funksjon som ikke var noe seriøst forsøk på å løse oppgaven, men da likevel passert de testene hvor lista ikke skulle endres. Dermed kan noen ha fått 2 korrekte tester med kode som ved manuell retting ville gitt null poeng, mens andre kan ha hatt kode som manuell retting ville gitt 4/5 men som ikke tilfredsstilte noen tester på grunn av en subtil liten feil.

For å oppsummere svar på forskningsspørsmålene: FS1) Hvordan fungerte «Kompilering» i Inspera sammenlignet med lignende oppgavetyper? Foreløpig dårligere enn for eksempel Moodle CodeRunner, både på grunn av en tungvint løsning med testtilfeller basert på input og print, med forhåndskode som alltid var synlig og editerbar for studentene, og på grunn av vag tilbakemelding til studentene ved feil testresultat. FS2) Hvordan var studentenes prestasjoner? Per deloppgave var det tendenser til at mange havnet på full score eller null score, kanskje også med ett mellomnivå (f.eks. 2/5) som mange kandidater presterte på. Aggregert over mange små oppgaver av typen «Kompilering» ble det imidlertid resultater med en jevn fordeling. Mange svake kandidater ble reddet av at iallfall 3 av deloppgavene var svært lette, samt at de kan ha scoret noen poeng på lette testtilfeller på de vanskeligere oppgavene. FS3) Hvilke forbedringer ønskes for denne oppgavetypen (i Inspera eller andre verktøy): Bedre løsning mhp testing og skjuling av testkode, samt bedre tilbakemelding til student, så programmeringen og testing blir mer autentisk – eller eventuelt bedre muligheter for integrasjon mellom verktøy for digital eksamen og industrielle programmeringsomgivelser, slik at

programmeringen på eksamen kan fremstå enda mer autentisk enn det en oppgavetype i et eksamenssystem har forutsetninger for å gjøre.

Samtidig er det viktig å understreke – som for eksempel indikert av analysen av oppgave 2g – at uansett hvilke forbedringer man får av verktøyet, er det selve oppgavedesignet som er aller mest avgjørende for om en vurdering blir vellykket. Noen tanker vi tar med oss videre for oppgavedesign i fremtiden kan være som følger:

#1) Vær forsiktig med testtilfeller som blir riktige for kode som ikke utretter noe som helst. Selv om det er relevant å teste f.eks. at ei liste ikke blir endret hvis den ikke skulle endres, må det unngås at slike testtilfeller får for stor vekt. (I oppgave 2g som vist over kunne man score 40% for kode som gjorde ingenting, dette er for mye).

#2) Prøv å legge opp oppgaven slik at testtilfellene blir gradvis vanskeligere å tilfredsstille, og helst slik at alle ulike poengscorer er plausible mellomstadium mellom null score og full score. I oppgavene vi så på, var det typisk flest studenter på full og null, samt kanskje på bare ett mellomstadium. Det var for eksempel sjelden å ha 4/5 riktige – som kunne ha blitt oppnådd hvis én test var vesentlig vanskeligere å tilfredsstille enn andre.

Alt i alt kan det sies at «Kompilering» - eller tilsvarende oppgavetyper med Moodle CodeRunner eller lignende – hvor man tenker å vurdere automatisk basert på antall passerte tester, er en utfordrende sjanger å lage oppgaver for, da man må tenke seg om vesentlig mer enn i en tradisjonell oppgave hvor studenten bare skal skrive koden og sensor skal vurdere manuelt. Dette tilsier at det kunne være interessant å ha samarbeid mellom faglærere ved mange universitet både i Norge og internasjonalt for å prøve å lage gode oppgaver til dette, enten man tenker å bruke dem formativt eller summativt.

References

1. Simon, Chinn, D., de Raadt, M., Philpott, A., Sheard, J., Laakso, M.-J., D'Souza, D., Skene, J., Carbone, A., Clear, T., Lister, R.: Introductory programming: examining the exams. In: Proceedings of the Fourteenth Australasian Computing Education Conference-Volume 123, pp. 61-70. Australian Computer Society, Inc., (2012)

2. Parsons, D., Haden, P.: Parson's programming puzzles: a fun and effective learning tool for first programming courses. In: Proceedings of the 8th Australasian Conference on Computing Education-Volume 52, pp. 157-163. Australian Computer Society, Inc., (2006)

3. Denny, P., Luxton-Reilly, A., Simon, B.: Evaluating a new exam question: Parsons problems. In: Proceedings of the fourth international workshop on computing education research, pp. 113-124. ACM, (2008)

4. Ericson, B.J., Margulieux, L.E., Rick, J.: Solving parsons problems versus fixing and writing code. In: Proceedings of the 17th Koli Calling International Conference on Computing Education Research, pp. 20-29. ACM, (2017)

5. Ihantola, P., Karavirta, V.: Two-dimensional parson's puzzles: The concept, tools, and first observations. Journal of Information Technology Education 10, 119-132 (2011)

6. Van Merriënboer, J.J., De Croock, M.B.: Strategies for computer-based programming instruction: Program completion vs. program generation. Journal of Educational Computing Research 8, 365-394 (1992)

7. Cheng, N., Harrington, B.: The Code Mangler: Evaluating coding ability without writing any code. In: Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, pp. 123-128. (2017)
8. Sindre, G.: Code Writing vs Code Completion Puzzles: Analyzing Questions in an E-exam. In: 2020 IEEE Frontiers in Education Conference (FIE), pp. 1-9. IEEE, (2020)
9. Harms, K.J., Rowlett, N., Kelleher, C.: Enabling independent learning of programming concepts through programming completion puzzles. In: 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), pp. 271-279. IEEE, (2015)
10.Lobb, R., Harlow, J.: Coderunner: A tool for assessing computer programming skills. ACM Inroads 7, 47-51 (2016)
11.Hamrick, J.B.: Creating and grading IPython/Jupyter notebook assignments with NbGrader. In: Proceedings of the 47th ACM Technical Symposium on Computing Science Education, pp. 242-242. (2016)
12.Gamage, S.H., Ayres, J.R., Behrend, M.B.: A systematic review on trends in using Moodle for teaching and learning. International Journal of STEM Education 9, 1-24 (2022)
13.Croft, D., England, M.: Computing with CodeRunner at Coventry University: Automated summative assessment of Python and C++ code. In: Proceedings of the 4th Conference on Computing Education Practice 2020, pp. 1-4. (2020)
14.Bralić, V., Kavran, K., Valić, B.: Automatic Programming Task Grading–A Case Study in CodeRunner use at the University of Applied Sciences Velika Gorica.
15.Pringuet, P., Friel, A., Vande Wiele, P.: CodeRunner: A Case Study of the Transition to Online Learning of a Java Programming Course. AUBH e-learning conference. American University of Bahrain, Bahrain (2021)
16.Blank, D.S., Bourgin, D., Brown, A., Bussonnier, M., Frederic, J., Granger, B., Griffiths, T.L., Hamrick, J., Kelley, K., Pacer, M.: nbgrader: A tool for creating and grading assignments in the Jupyter Notebook. The Journal of Open Source Education 2, (2019)
17.Zhao, P., Xia, J.: Use JupyterHub to Enhance the Teaching and Learning Efficiency of Programming Related Courses. 23-ICIT. BNU/HKBU-UIC (2019)
18.Manzoor, H., Naik, A., Shaffer, C.A., North, C., Edwards, S.H.: Auto-grading jupyter notebooks. In: Proceedings of the 51st ACM Technical Symposium on Computer Science Education, pp. 1139-1144. (2020)
19.Sindre, G.: Analyse av studenters eksamenskode: typiske feil og muligheter for autoretting. UDIT, (2020)