Developing and Evaluating LLM Assistants in Introductory Computer Science

Laryn Qi

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-132
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-132.html

May 17, 2024

Developing and Evaluating LLM Assistants in Introductory Computer Science

by

Laryn Qi

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Narges Norouzi Co-chair
Professor John DeNero, Co-chair

Spring 2024

Developing and Evaluating LLM Assistants in Introductory Computer Science

by Laryn Qi

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Narges Norouzi

Professor Narges Norouzi
Research Advisor

5/17/2024

(Date)

* * * * * * *

Professor John DeNero
Research Advisor

5/16/2024

(Date)

Developing and Evaluating LLM Assistants in Introductory Computer Science

Abstract

Developing and Evaluating LLM Assistants in Introductory Computer Science

by

Laryn Qi

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Narges Norouzi, Co-chair

Professor John DeNero, Co-chair

Chatbot interfaces for LLMs enable students to get immediate, interactive help on homework assignments, but doing so naively may not serve pedagogical goals. In this thesis, we first report on the development and deployment of an LLM-based homework assistant for students in a large introductory computer science course. Our assistant offers both a "Get Help" button within a popular code editor, as well as a "Get Feedback" feature within our command line autograder, wrapping student code in a custom prompt that supports our pedagogical goals and avoids providing solutions directly. We explore class-wide effects of deploying this AI assistant, showing that hints are appreciated by students, and the bot's effects on reducing homework completion time are concentrated among students with above-median times, suggesting that these hints can help struggling students make more rapid progress on assignments. Then, we present and evaluate **three research questions that measure the effectiveness of the assistant's hints**: *Do the hints help students make progress? How effectively do hints capture issues in their code? Are students able to apply the hints to make progress?* Through analysis of **knowledge components**, we show that students are able to resolve problems with their code and approach a working solution more quickly with access to hints, that the assistant's hints are able to consistently capture the most pressing errors in students' code, and that hints that address a few issues at once rather than a single bug are more likely to lead to student progress. This thesis is an extension of [40].

To Ma, Ba, Ge, and Jie. And to Lucy.

Contents

List of Figures

List of Tables

Acknowledgments

To JD, you showed me what it takes to be a great researcher. Each of our regular debugging deep dives and statistics sessions (at *any* hour of the day) was an invaluable learning experience. Thank you for your mentorship and for your infinite patience with me.

To Narges, you helped me find my bearings in this unfamiliar and daunting landscape of CS education research. The endless wealth of new ideas you brought to each meeting guided my every experiment. Thank you for pushing me to be the best researcher I could be and for taking a chance on me.

To John, in my first semester at Berkeley, you taught me computer science. Then throughout undergrad, you taught me how to teach others computer science. And in grad school, you taught me how to teach an AI to teach others computer science. You are truly a master at the art of simplifying a problem down to its most essential form. Thank you for always placing your trust in me and for giving me direction when I was most lost.

I extend my sincerest gratitude to you all.

And to all my professors, TAs, colleagues, students, and friends whom I've had the fortune of learning from at Berkeley – I thank you. I will forever cherish these years.

Joint Work

Chapters 1-3 are adapted from [40] and were primarily authored by J.D. Zamfirescu-Pereira and Narges Norouzi.

Chapter 1

Introduction

The recent wide availability of ChatGPT and similar Large Language Models (LLMs) has given students in introductory Computer Science (CS) courses a tempting alternative to asking for help on programming assignments — and potentially waiting hours to receive it. However, while naively-used LLMs do help students solve assigned problems, typically by providing them with correct answers along with explanations, such use of LLMs allows students to avoid the process of developing solutions themselves and the learning associated with this process.

In 61A, the introductory CS course at UC Berkeley, we deploy our tool called **61A Bot**. It identifies the problem the student is working on, collects their code, and wraps these in a custom prompt for GPT-4 [28]. We designed this prompt to steer GPT-4 towards feedback that mirrors how we ourselves typically approach student questions: identifying whether the student understands the question, which concepts students might need reinforcement on, and whether they have a plan, and then helping students by providing conceptual, debugging, or planning support as appropriate.

In this paper, we first give an overview of this tool's development & deployment. Then, we present a few holistic, course-wide findings. Finally, we dive deeper into student assignment progression and quantitatively evaluate the pedagogical effects of LLM-generated hints by posing 3 research questions (RQs):

RQ1: *Do the hints help students make progress?*

RQ2: *How effectively do the hints capture issues in the code?*

RQ3: *Are students able to apply the hints to make progress?*

Chapter 2

Background & Related Work

Generative models such as ChatGPT,[1] OpenAI Codex [2], DeepMind AlphaCode [19], Amazon CodeWhisperer, and GitHub Copilot[2] offer promising opportunities for enriching the learning experience of students. These models have already been leveraged by educators in different areas of Computing Education [12, 7, 13, 6, 32], where they accelerate content generation and seem to be impacting the relevant skills students gain in introductory CS courses. Researchers have studied LLMs in areas such as generating code explanations [17, 1, 21, 9], providing personalized immediate feedback, enhancing programming error messages [18, 36], generating discussion forum responses [25, 20], and automatic creation of personalized programming exercises and tutorials [34, 39, 33] to enhance the comprehensiveness of course materials.

However, the integration of LLMs in introductory CS instruction comes with challenges. Students could become overly reliant on automation (a concern at least as old as calculators [5]), potentially hindering their development of critical problem-solving skills — though recent work suggests such hindrance is not inevitable for programming assistance [15]. Taken to an extreme, the resulting absence of human interaction could have negative effects, alongside other ethical concerns related to plagiarism and the responsible use of LLM-generated code. To maximize the benefits of LLMs while mitigating these challenges, a thoughtful and balanced approach to their incorporation into introductory CS courses is essential [8, 24, 22].

By deploying LLMs as intelligent programming assistants, students can receive immediate, personalized support and guidance, fostering a deeper understanding of coding concepts and promoting self-paced learning — just as did pre-LLM Intelligent Tutoring Systems (see [4] for a review, and [35] for a specific example). However, the ability of LLMs to generate *tailored* resources, such as tutorials and code examples, not only expands the available learning materials but also accommodates students' varying learning styles and preferences — though these generated materials are not always better [29]. Educators should integrate LLMs as *complementary* tools, striking a balance between automation and human interaction, while emphasizing the development of critical problem-solving skills and responsible

[1] `https://chat.openai.com/`

[2] `https://copilot.github.com/`

coding practices, ultimately serving students better in their CS education. Researchers are increasingly utilizing LLMs as chatbots in courses [9, 37] or online educational websites [27] to provide immediate personalized feedback, human-AI pAIr programming paradigm [38], or tools in supporting students' development of programming skills [31, 11, 3]. This paper contributes to the body of educational research and pedagogical innovation, demonstrating the transformative potential of technology-driven approaches in reshaping how CS fundamentals are taught and learned.

Chapter 3

Method: Notes on Design & Deployment

We considered a number of ways to expose LLM capabilities to students, including varying interaction modes (e.g., chat conversations, Q&A, one-shot requests) and support modes (e.g., debugging, conceptual scaffolding, student assessment). Though there are many ways LLMs seem likely to impact early programming instruction, we chose to address one of the more challenging bottlenecks we faced in our large course at a large public institution: the availability of tutors and other staff to help students when they get stuck on homework (HW) problems. In addressing this specific challenge, we chose early on to focus specifically on debugging homework assignments.

One early decision with pedagogical implications was whether to support students *responding* to the assistant in natural language. Three concerns, about hallucinations, about students sharing personal information with a third party, and about what harms might come from an unmonitored chat interaction, led us to deploy a one-click "Get Help" interaction mode without an opportunity for follow-up. This meant that one valuable pedagogical ingredient — having students explain their understanding of the problem — would remain out of reach in this deployment.

Our development process was based on the set of homework questions assigned throughout our course, author-generated constructions of incomplete code, and a set of student checkpoints — also incomplete code, often containing errors — collected over the prior year. We developed our system by testing our prompts-under-development on these homework problems using a custom-built prompt engineering system (see Figure 3.1).

Initial evaluations of GPT-4 on a small set of typical introductory CS questions used in our course in prior years suggested that GPT-4 could provide effective support across many avenues, including debugging — and was much more effective than GPT-3.5 and other LLMs, which focused our tuning efforts on prompt engineering over fine-tuning and other methods. Following a common tutoring pattern, we designed a prompt that would try to assess student conceptual knowledge, based on the provided code, and offer syntactical, logical, or even template-code suggestions — but not solutions. This prompt explicitly includes a sequence of steps to consider in response to student code, modeled in part on our own personal tutoring processes:

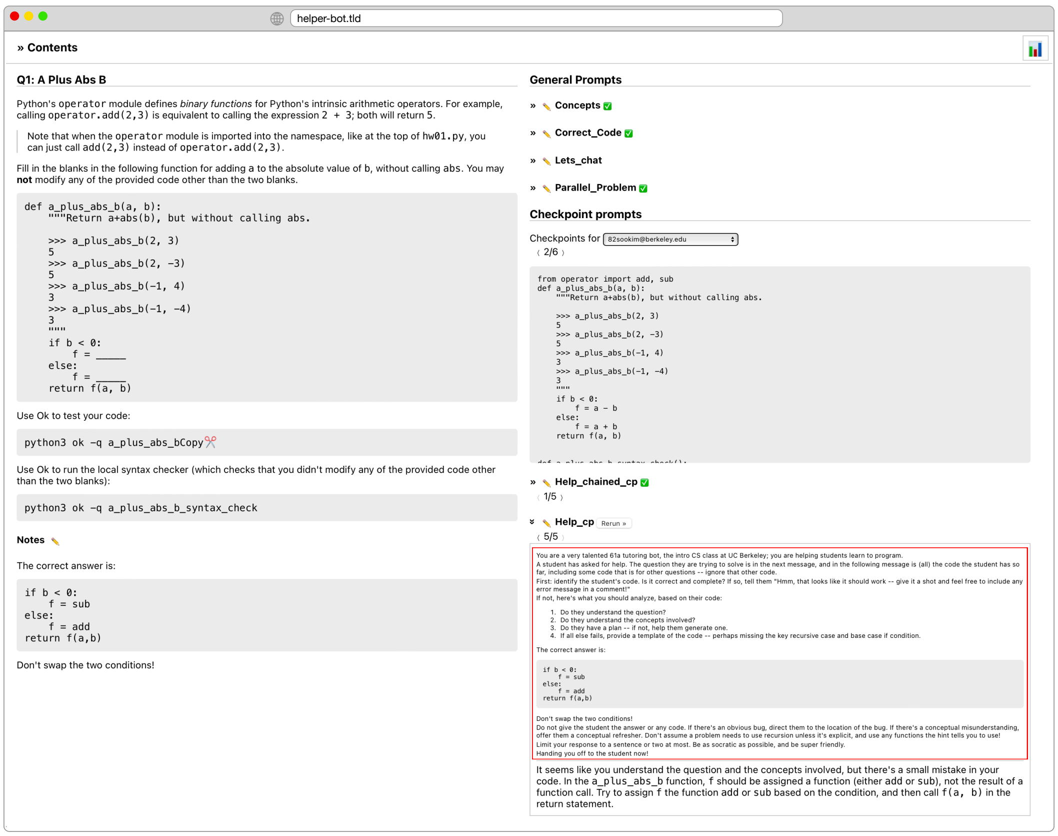

Figure 3.1: Screenshot of our custom prompt engineering tool. On the left, a specific homework assignment question, with a manually-authored "Notes" containing the correct solution and a pointer not to swap conditions; on the right, a set of prompts, checkpoints, and outputs. Here, a prior year student's incorrect code is loaded under "Checkpoint prompts", the **Help_cp** prompt is activated (red outline), and the output is displayed below the prompt text.

```
1) Is the student missing any conceptual knowledge? Would a refresher help?
2) Is the code they have already, if any, on the right track?
3) How close are they to a working solution?
4) Did they follow your previous instructions? If not, rephrase and offer
   advice in a different way.
5) Do they have a plan -- if not, help them generate one.
6) If all else fails, provide a template of the code -- perhaps missing the
   key recursive case and base case if condition.
```

Though we avoided students explicitly writing natural language "chat" messages to the bot, we did want some degree of continuity — which we achieved by also including up to three prior *student code - bot advice* exchanges if available (step 4).

In addition to the steps above, the prompt also includes instructions such as `Do not give the student the answer or any code` and `Limit your response to a sentence or two at most`. An earlier draft version of this prompt appears in the red outline in Figure 3.1, with the complete deployed prompt reproduced in Appendix A.1.

3.1 Development & Prompt Engineering Challenges

From the literature, we expected that trying to engineer dialogues through an LLM would be challenging in ways that would be hard to address [41], and indeed we found this to be true for us as well. Early on, in testing extended dialogue interactions, we found an increasing likelihood that GPT-4 would provide a direct solution as the conversation extended — validating our decision to start with a single-shot request rather than dialogue.

Many initial challenges were addressable through prompt changes alone: for example, by including the instruction `Don't assume a problem needs to use recursion unless it's explicit`, we avoided spurious suggestions that loop-based code should be rewritten using recursion.

But one particularly frustrating challenge illustrates a number of the failures we observed early on: an over-eagerness to "correct" student code that was, in fact, already correct. One such homework problem asked students to fill in a template to compute $a + |b|$, using the operators `add` and `sub` based on whether $b < 0$ (see Figure 3.2). With an early version of our prompt, GPT-4 would consistently question the correct solution, asking students to think about what should be done in the case $b < 0$.

In addressing this and other similar issues, we experimented with asking GPT-4 to first generate a complete, correct solution, and then copy the student's code, reasoning that having these within the prompt response would reduce the likelihood that GPT-4 would incorrectly label correct code as containing errors. Unfortunately, this approach introduced substantial additional latency that made us decide against deploying it: in many cases, it added several hundred tokens to the output — none of which could be shown to students in a streaming fashion — increasing time-to-response by tens of seconds, an unacceptable trade-off given our design goal of rapid interactive feedback.

Hoping to find a prompt that would prove robust to variations in course content, we initially avoided any problem-specific text beyond a statement of the problem itself. However, we ultimately added a problem-specific note for those homework problems that required extra steering. In the *operator* assignment above, this note contains a solution to the problem.

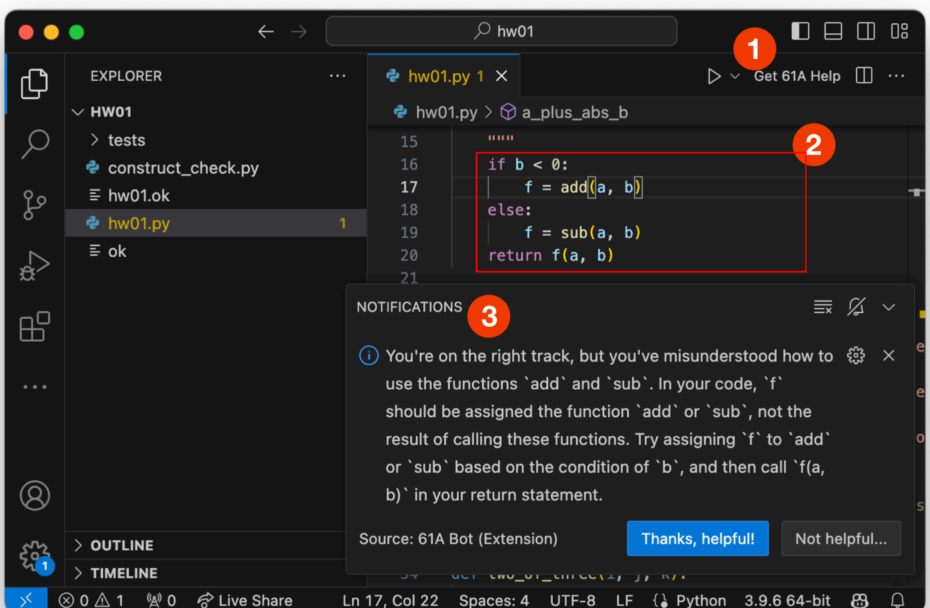

Figure 3.2: Screenshot of the VS Code user interface as students see it. Note the “Get [redacted] Help” button in the toolbar at the upper right-hand side **(1)**, which students click to send their code to the helper bot. Bot replies appear in a pop-up notification (dialog window **(3)** with [Thanks, helpful!] and [Not helpful...] buttons) containing the bot's advice on the student's current code (**(2)**, red outline)—in this case, a helpful suggestion that the student shouldn't *call* the functions `add` and `sub`, but merely assign them to variable `f`.

3.2 Deployment

We deploy our tool in two modalities. In one modality, students click on a "Get Help" button (see Figure 3.2) in the VS Code[1] editor toolbar to activate our 61A Bot extension. The extension collects students' code, makes a best guess of which homework problem the student is working on (as several problems often appear in a single file), and constructs a request. In the second modality, students run an autograder[2] in the command line which collects the student's code, any errors from executing test cases, and constructs a request from these (shown in Figure 3.3). These requests reach a server run by our instructional staff, which wraps the student data in our prompt, passes it along to GPT-4, and logs the request and GPT-4's response for further analysis.

Students are informed when installing the software[3] that, in using our assistant, all code they write will be sent to OpenAI via Microsoft Azure [23], and that they should not include any content in their code files (e.g., comments) that they are not comfortable sharing.

In Fall 2023, we initially deployed our helper bot VS Code extension to an experimental section of 400 students a few days into the start of our academic year, followed by a full-scale deployment in week 7 for the approximately 1300 students across both course sections. The students in the initial experimental section were not randomly selected, but rather were drawn primarily from a pool with a stronger academic background in math and computing. In conjunction with the wider deployment, we also enabled access through the autograder tool students could already run from the command line to validate their code against a set of test cases. In Spring 2024, we deployed both the VS Code extension and command line integration immediately from the start of the term to all 872 enrolled students.

[1] Visual Studio Code, `https://code.visualstudio.com`
[2] OK Client, `https://github.com/okpy/ok-client`
[3] 61A Bot Installation Instructions, `https://cs61a.org/articles/61a-bot/`

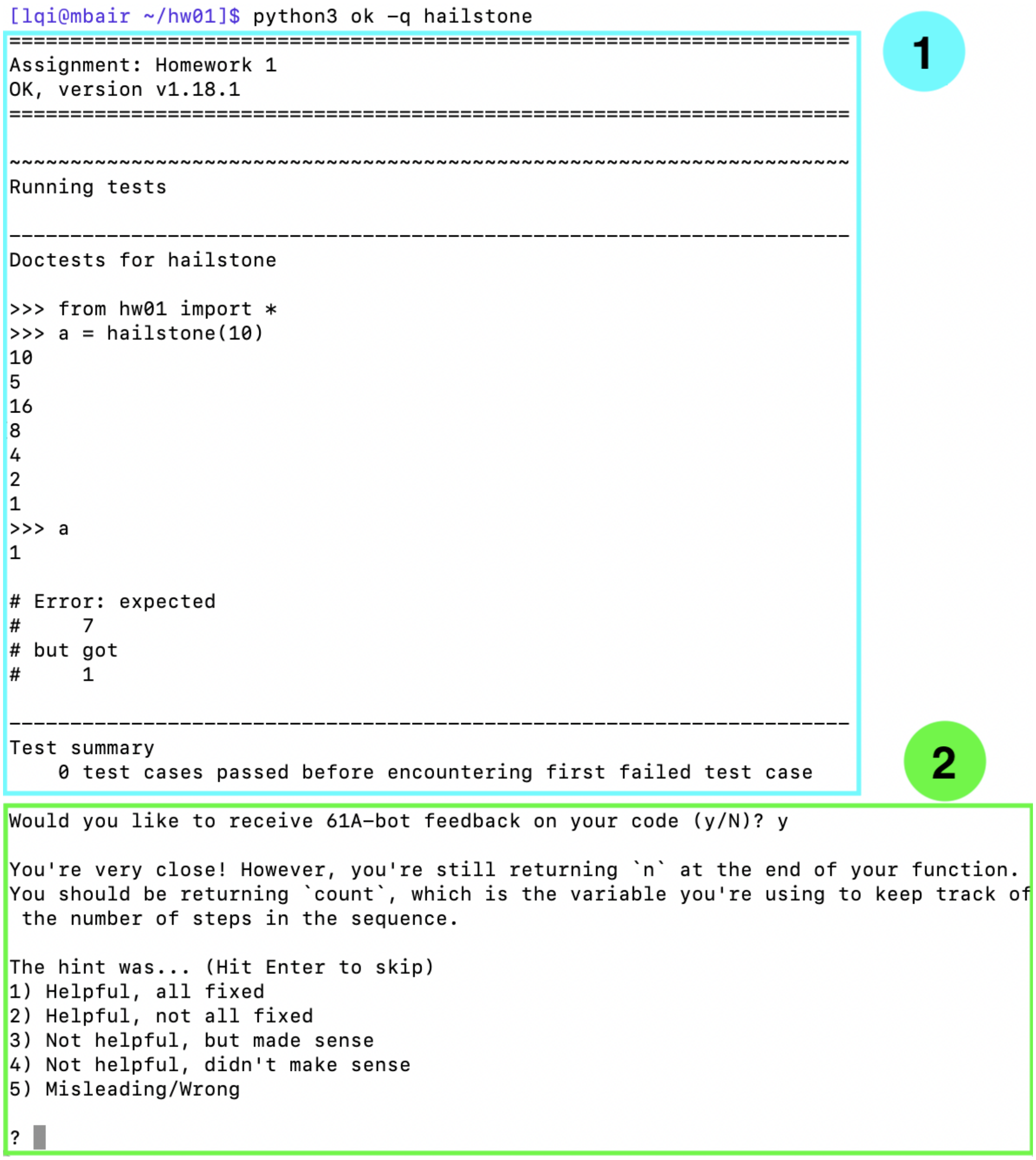

Figure 3.3: Screenshot of second modality of 61A Bot deployment: command line integration. (1) indicates the existing autograder feedback students receive. (2) shows the extended student workflow of requesting a hint from 61A Bot.

Chapter 4

Findings

Here, we present a number of aggregate results of deploying an AI assistant in 61A.

4.1 Usage

Usage patterns suggest that students are finding the bot helpful, returning to it multiple times as they engage in homework (see Figure 4.1). As expected, usage increases as the assignment deadline nears, and is concentrated in the late afternoons and evenings.

Comparing usage between the two modalities, over 93% of 61A Bot queries were made via the command line instead of VS Code. This stark contrast is expected as the "Get Feedback" option is provided by default in the workflow of the command line autograder, which students have always ran liberally and must download with each homework assignment. On the other hand, the "Get Help" button in VS Code requires a manual installation and sometimes outputs less helpful hints as these queries do not have access to autograder test case outputs.

Engagement with the assistant seems to be dependent in part on the types of errors student encounter and the feedback they receive. Anecdotally, syntax errors and function misuse (when correctly identified by the bot) are commonly resolved with a single request, while conceptual misunderstandings are unsurprisingly much trickier; a common pattern for the latter is a student repeatedly requesting help while the bot responds with minor variations on the same message, until finally through GPT-4's stochasticity a response contains a key new component that enables the student to progress (see Appendix B.2). (A few weeks into the initial deployment in Fall 2023, we added the “prior context” functionality described earlier; this helped substantially with response diversity, but helped students less than expected — in part because it is not always clear *why* the previous bot response was inadequate).

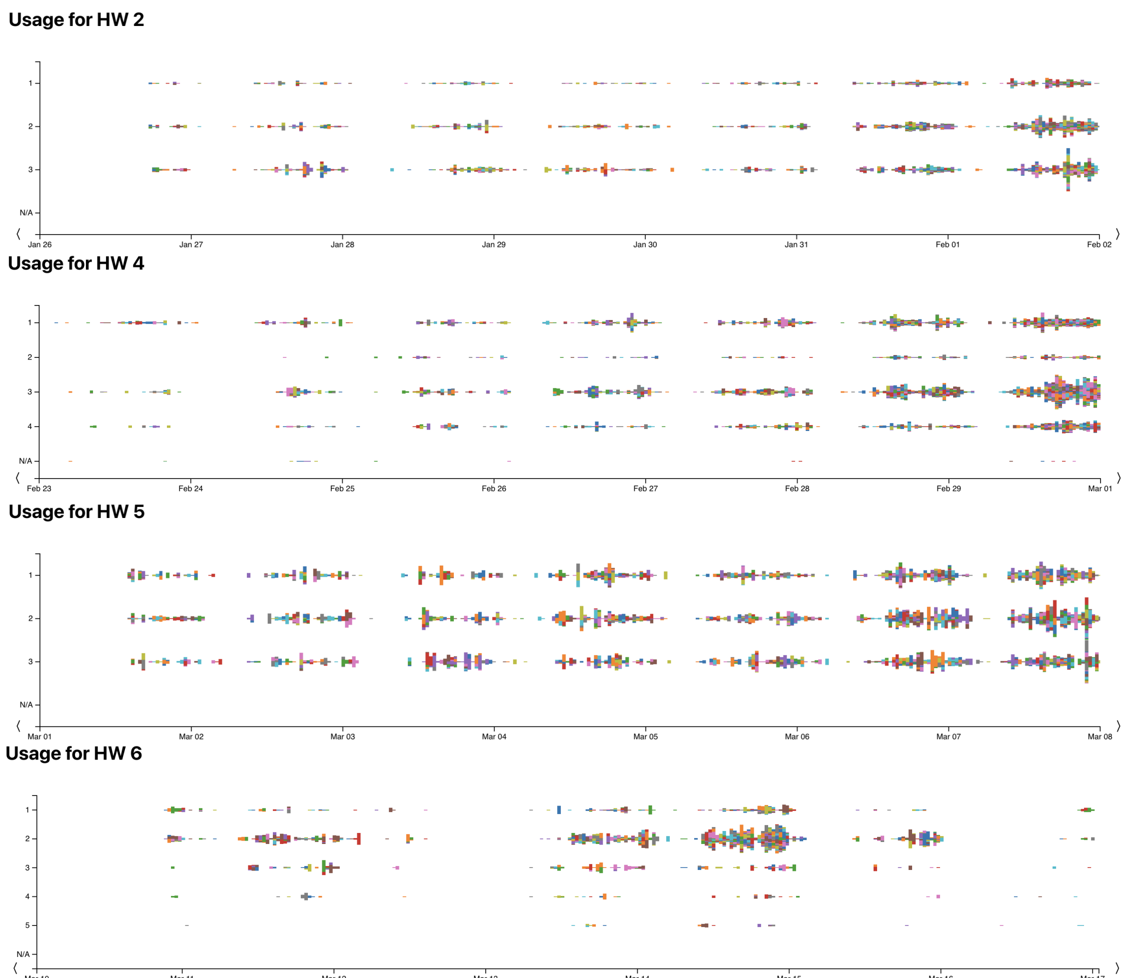

Figure 4.1: Usage patterns across four weeks of homework in Sp24; x-axis ticks mark midnight on the days specified; y-axis ticks identify specific questions within the week's assignment, with "N/A" indicating that no specific question was identified. Requests are binned into time blocks; bar y-height represents the number of requests in each block, scaled to the row height. Student IDs are binned into ten different colors for visibility.

4.2 Successes & Failures

Overall, repeated use of our assistant by individual students, alongside anecdotal reporting and explicit [Thanks, helpful!] feedback, suggests that students derive at least some value from having access to the assistant.

On the "successes" side, we found GPT-4's conceptual explanations quite strong; consider this explanation of the recursive step and the base case in a coin-counting problem:

> Consider this: for each coin, you can either include it or exclude it. If you include it, you subtract its value from the total and recurse. If you exclude it, you move to the next coin and recurse.

...and...

> The base case of your recursion would be when the remaining change is 0, in which case you have found a valid way to make a change, so you should return 1. If the remaining change is less than 0 or there are no more coins to use (i.e., the current coin is None), you should return 0 because there are no valid ways to make change in these cases.

Even in cases where multiple errors were present in student code, a single response enumerating them could provide a lot of value at once, in a format that allowed students to re-read as they worked on their code — an unexpected benefit over traditional spoken 1:1 tutoring. This phenomenon is further concurred in §5.3.

On the "failures" side, the predominant error modes could be characterized as **hints that students apply incorrectly** or **hints that students do not apply**. We typically observe the former case when the student has an understanding of the hint that is incomplete or otherwise inaccurate, leading to wrong applications of the instructions in the hint as seen in Appendix B.1. As a result, in these cases, 61A Bot often has to try and keep up with new bugs arising in each subsequent code submission due to the previous incorrect application of a hint. In the latter scenario, students often ignore the hint outright, either making unrelated changes or no changes at all. Sequences of submissions in this mode usually lead to the same hint being repeated in different ways until the student makes progress (shown in Appendix B.2).

4.3 Student Reception

In Fall 2023, we conducted a non-anonymous survey at the end of the semester where 698/1407 students (49%) students responded. Students were asked to rate how much they used the bot once they discovered it and how helpful they found it on a scale from 1-5. A summary of the results can be found in Table 4.1. As expected, those who reported more usage also found it more helpful.

Count	*How helpful was the 61A Bot?* [1]							
How much did you use the 61A Bot once you discovered it? [2]	**N/A**	**1**	**2**	**3**	**4**	**5**	**Grand Total**	**Avg. Rating**
N/A	55	0	0	0	0	0	55	
1	21	43	22	26	2	0	114	1.9
2	1	7	35	31	11	2	87	2.6
3	0	6	17	57	41	14	135	3.3
4	0	2	13	40	81	32	168	3.8
5	0	0	4	19	36	80	139	4.4
Grand Total	**77**	**58**	**91**	**173**	**171**	**128**	**698**	3.4
3, 4, or 5		8	34	116	158	126	442	3.8
%		1.8	7.7	26.2	35.7	28.5	100.0	

[1] On a scale from 1 "Not helpful" to 5 "Very helpful"
[2] On a scale from 1 "Not a lot" to 5 "A lot"

Table 4.1: Fall 2023 End-of-semester 61A Bot Survey Results

In Spring 2024, we conducted a non-anonymous survey at the end of the semester where 774/872 (89%) students responded. On a scale from 1-5, we asked students to rate their own bot usage, bot helpfulness, bot reliability, and overall satisfaction with the bot. Finally, we asked them whether or not they recommend that the bot be available to students in future semesters. A summary of these survey results are found in Table 4.2.

	Average Rating
Usage	3.31
Helpfulness	3.66
Reliability	3.40
Satisfaction	3.74
Would Recommend? (Yes/No)	**93%** Yes

"Usage", "Helpfulness", "Reliability", and "Satisfaction" were each rated on a scale from 1-5 with
1 being "No usage", "Not helpful", "Not reliable", and "No Satisfaction" and
5 being "A lot of usage", "Very helpful", "Very reliable", and "Very satisfied".

Table 4.2: Spring 2024 End-of-semester 61A Bot Survey Results

In the Spring 2024 survey, we also asked two optional questions: what did students like the most and least about learning with the bot?

The following is a small sample of responses to both questions that capture the general student sentiment:

<u>What students liked the most</u>:

> "What I liked the most was that it made assistance much more accessible as a commuter. It would cut down the hours I would have to spend in office hours a week and gave me the confidence that I could complete the homework on my own."

> "What I loved about the bot is that it allowed me to get feedback when I didn't have access to a tutor. Accordingly, instead of banging my head against the wall for hours, I was able to get feedback about what I was doing wrong and correct the mistakes. For me personally, I would have had a lot more success in this class if I would have had access to the bot for labs. (Labs on average took me about 2 and a half hours to complete, and sometimes longer if I didn't have access to a tutor)."

> "The bot was definitely very helpful in helping me start the problems and at times, debug my code and get me back on track. I felt that it introduced me to methods to solving problems I never thought of before when programming a solution."

> "I loved 61a bot because I was determined not to use ChatGPT. Instead 61a bot was the perfect in between as it helped me understand why my logic was wrong and gave me hints on how to fix it."
>
> "Really fast feedback time. Sometimes waiting until office hours to ask something or help debug can be frustrating, especially when it ends up being a silly error that the bot can easily help with in a few seconds."

<u>What students liked the least</u>:

> "Sometimes the answers were slightly vague. Of course, the bot can't simply spit out the answer, but sometimes it was frustrating how it would say "you're on the right track, but there seems to be a conceptual misunderstanding with __" — the explanations for the blank could be a bit jargon-filled and didn't always directly help me resolve the misunderstanding due to imprecise language."
>
> "Sometimes it explains questions and gives hints of what to fix, but does not help my understanding of the question."
>
> "It would repeat the same things over and over again."
>
> "It keeps popping up after every question."
>
> "I think the bot tended to get confused and give false corrects with Scheme assignments the most."

In summary, students generally appreciated the accessibility, debugging capabilities, and time saved from the bot. On the other hand, the hints would sometimes be too vague for the students to make changes while other times the bot would be too specific and give away too much. To this end, we plan to make hints more personalized in future semesters (see Next Steps). In some cases, the bot would repeat the same hints. Despite us fiddling with prompt changes to encourage rephrasing and diversity in responses to repeat inputs, these incidents still occur — it's a difficult tradeoff to balance as we don't want to risk more hallucinations by upping the temperature, for example. Some students dislike the disruption of the traditional 61A command line workflow (pre-61A Bot), so we plan to make the hint pop-up optional in the autograder setting. Lastly, the bot sometimes gives outright false advice, specifically for Scheme problems involving many parentheses. Presumably, GPT-4 has trained on far less Scheme than Python code, so this shortcoming is expected. One remedy is to include additional few-shot expert examples in the prompts for Scheme assignments in particular.

4.4 Reduction in Completion Time

One way of quantitatively measuring the bot's effect is looking at how long students spend on homeworks. In 61A, each time the student runs the autograder, their submission gets logged on our server[1] as a checkpoint. So, we can estimate the total time spent on an assignment by summing up the time deltas between consecutive checkpoints. To account for students completing the homework across multiple sessions, we ignore time deltas above a 30 minute threshold — a value chosen somewhat arbitrarily, however, we have confirmed that the results we report here are robust in the range of threshold values from 20 to 120 minutes.

We calculate these completion time estimates for a number of Python homework problems that are shared across multiple semesters. Although we initially did not see a statistically significant difference in median completion times, when we plot the completion time cumulative distribution functions (CDFs), we observe a clear difference in the curves for Spring 2024 when stacked against all other recent semesters (see Figure 4.2). While fixing an x-coordinate then looking at the vertical difference in densities may suggest a marginal effect, instead fixing a y-coordinate and looking at the *horizontal* difference in time spent along the x-axis reveals the significance in time decrease between Spring 2024 and past semesters, specifically in the 60th-80th percentile regime nearly across the board for the selected problems (up to 20 minutes in the case of `count_coins`). This implies that, while the fastest 60% and slowest 20% students haven't been affected consistently by the introduction of the bot, an above-median 20% of students are completing homework problems both significantly and consistently faster since the introduction of the bot (see also Figure 4.3 and Appendix C.1 for additional examples). For the specific group of struggling students in the 60th-80th percentile range that benefit immensely, it's feasibly the case that they have an imperfect understanding of the material which 61A Bot is able to identify and correct for the student. However, for students who have a stronger grasp on the material (in the 60th or lower percentile of completion time), it may be the case that the bot doesn't affect their workflow significantly or these students outright ignore the bot's hints. Furthermore, the students spending a very long time (in the 80+th percentile) on the homeworks may not be able to digest and successfully apply the bot's hints to make progress more quickly.

In search of evidence that this reduction in completion time wasn't merely a result of Spring 2024's student cohort being abnormally fast at solving 61A problems, we also looked at *lab* completion times. In 61A, lab assignments are similar in format and scope to homework assignments, with the primary difference being that our 61A Bot was **only enabled in homework assignments**, not lab assignments. In Figure 4.4, we observe that the distribution of lab completion times early on in the term (weeks 1 & 2) is nearly identical across semesters, suggesting students come in with a similar level of ability across terms. Subsequently, we look at early homeworks in Figure 4.5 and see that students are finishing homeworks significantly more quickly. These assignment-level CDFs corroborate the ob-

[1] OK Server, `https://github.com/okpy/ok`

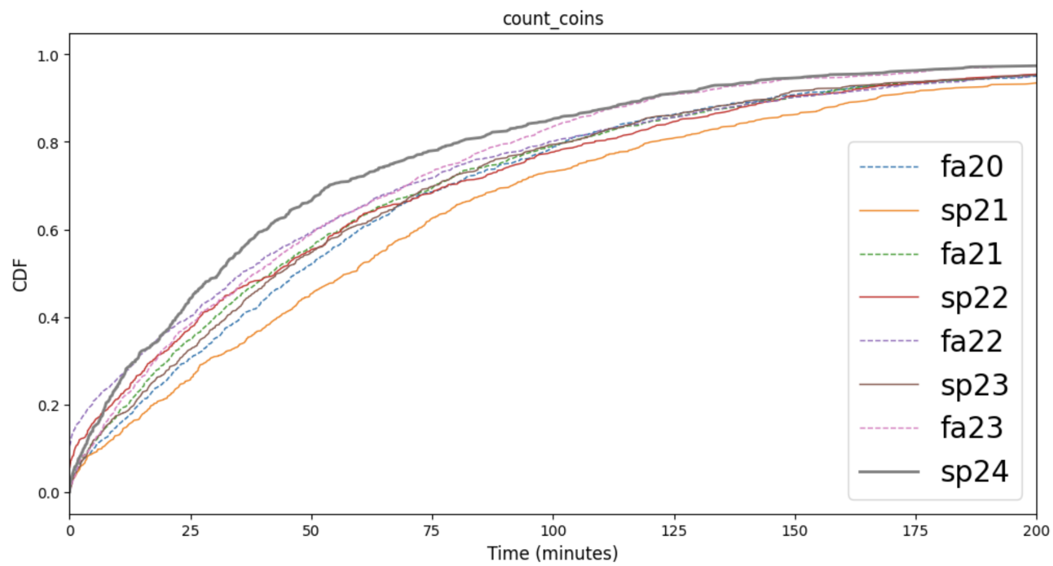

Figure 4.2: `count_coins` CDFs

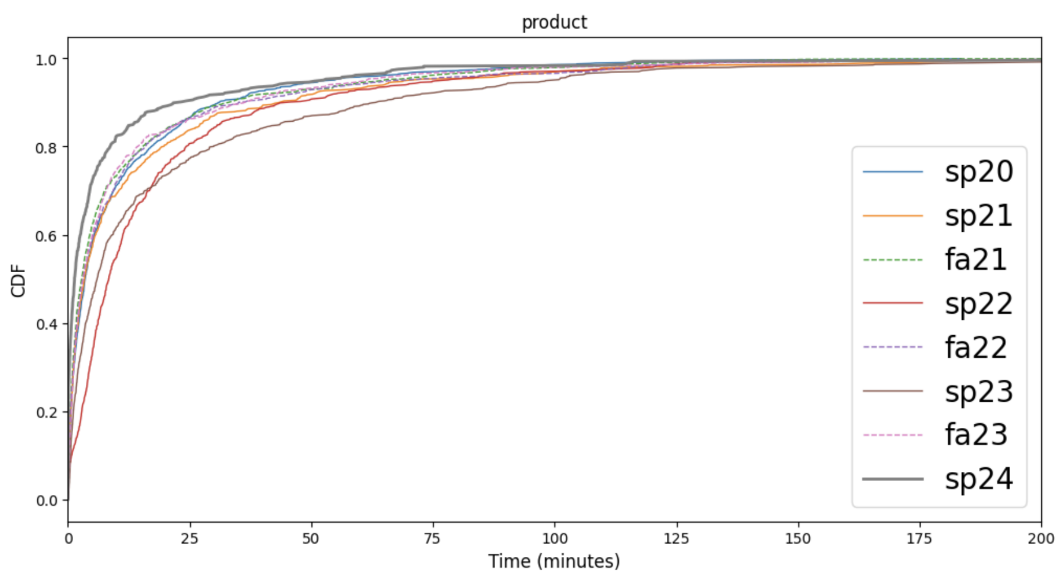

Figure 4.3: `product` CDFs

servation from the previous problem-level CDFs in Figures 4.2 & 4.3 that the 60th-80th percentile of students gain the most benefit from our tool. To further validate our hypothesis about Spring 2024 homework completion times, we performed pairwise completion time distribution comparisons across the Fall 2022, Spring 2023, Fall 2023, and Spring 2024 semesters for homeworks 1 & 2 (shown in Appendix C.2) where Spring 2024 is the only semester with universal 61A Bot access. We observe the following consistently:

- Fall 2022 and Fall 2023 have similar distributions.
- Spring 2023 students tend to take longer than Fall 2022 and Fall 2023 students, especially in the 60th-80th percentile.
- Spring 2024 students tend to finish faster than Fall 2022, Spring 2023, and Fall 2023 students, especially in the 60th-80th percentile.

Excluding Spring 2024, the fall-spring discrepancy is expected as the fall cohorts tend to include a higher proportion of CS majors while spring cohorts tend to have a greater diversity of majors. This difference serves to further highlight the anomaly of Spring 2024 completion times. These assignment-level plots also include probability density function (PDF) curves as an alternative view of the completion time distributions.

However, when we look later into the semesters, we actually start to see labs taking longer in Spring 2024 compared to Spring 2023 (see Figure 4.6). This suggests that over-reliance on the bot in the homework assignments might be a real issue leading to greater completion times on assignments without bot access, which aligns with the student feedback (shown previously in § 4.3) asking for 61A Bot to be incorporated in labs. Short-term fixes to this problem include implementing throttling as similar tools at Harvard [20] do. The ideal long-term solution to this problem would maintain unrestricted access to bot and improve hint quality to ensure the hints are helping students *learn concepts* rather than being used as a naive debugging tool (see Next Steps).

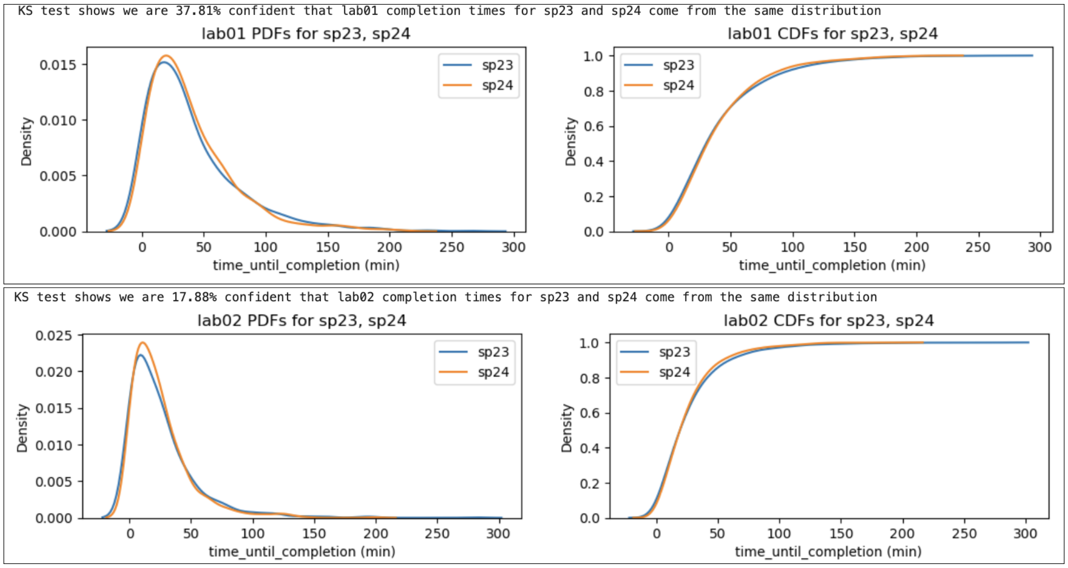

Figure 4.4: Spring 2023 vs Spring 2024 Labs 1 & 2 Completion Time Distributions (note: any density assigned to negative completion time is due to smoothing)

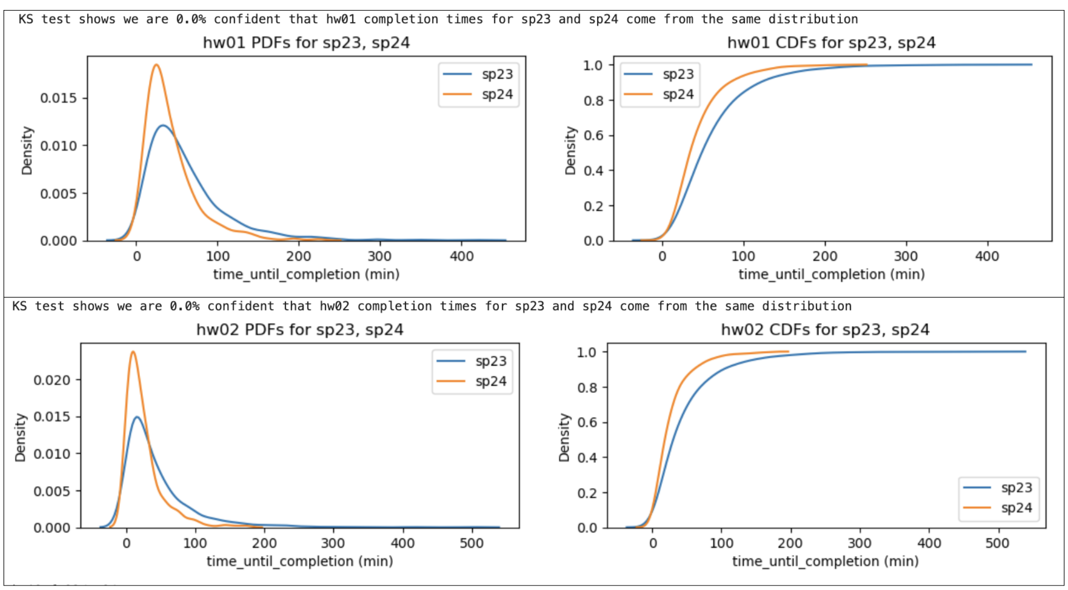

Figure 4.5: Sp. 2023 vs Sp. 2024 HWs 1 & 2 Completion Time Distributions (note: any density assigned to negative completion time is due to smoothing)

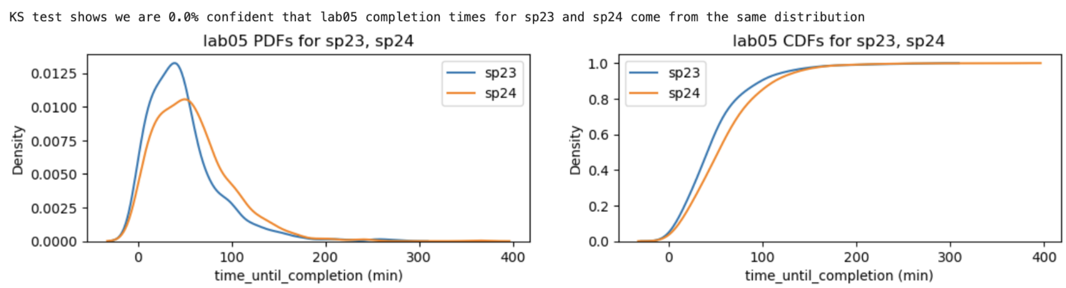

Figure 4.6: Spring 2023 vs Spring 2024 Lab 5 Completion Time Distributions (note: any density assigned to negative completion time is due to smoothing)

4.5 Reduction in Staff Reliance

Another reduction we observe is on Ed [10], a third-party platform that is used as an online discussion forum for students to receive asynchronous help. There is a dramatic near-50% decrease from 2900 to 1500 in the number of questions asked throughout the semester in Spring 2024 compared to Spring 2023 (see Appendix C.3). When we normalize by number of enrolled students (Table 4.3) and filter specifically for homework questions (Table 4.4), this effect persists. From Fall 2022 to Fall 2023, we observe striking consistency in the number of questions asked per student, when comparing the periods before the bot was widely introduced (late in the Fall 2023 semester). However, from Spring 2023 to Spring 2024 (2023 being before our 61A Bot deployment, while in 2024 access to 61A Bot was universal), we observe a significant decrease — a feasible byproduct of introducing the 61A Bot widely. Running a t-test between Fall 2022 and Fall 2023 yields $p = 0.245$ while a t-test between Spring 2023 and Spring 2024 yields $p = 0.003$.

	# questions	**# enrolled**	**Avg. # questions per student**
Fa22	3177	1656	1.918
Sp23	2888	1169	2.470
Fa23	2759	1407	1.961
Sp24	1478	872	1.695

Table 4.3: Average number of questions asked on Ed per student in Falls 2022, 2023 & Springs 2023, 2024

	# *HW* questions	**# enrolled**	**Avg. # *HW* questions per student**
Fa22	391	1656	0.236
Sp23	569	1169	0.487
Fa23	306	1407	0.217
Sp24	101	872	0.116

Table 4.4: Average number of *homework* questions asked on Ed per student in Falls 2022, 2023 & Springs 2023, 2024

Chapter 5

Results

As a framework for evaluating the 3 RQs presented previously in §1, we ground our analysis in **knowledge components** (KCs), first introduced by Koedinger et al. [16]. KCs represent course topics that students gain mastery of by completing assignments. They provide a finer-grained metric of student progression throughout an assignment which we use to measure students' abilities to grok and apply 61A Bot's hints as well as 61A Bot's ability to address relevant issues in students' understanding. Previous work [26] has shown the capacity of state of the art (SOTA) LLMs to effectively extract KCs from student code in CS1. We make use of this capability and extend it to also extract KCs from the hints generated by 61A Bot.

5.1 RQ1: Do the hints help students make progress?

First, for each student, we find which KCs they're missing as they progress through a homework problem by looking at their code checkpoints. We pass the student's code, a list of human-generated KCs (see Appendix D), and some metadata (see Appendix A.2) to GPT-4 and ask it to determine which KCs the student is missing from the provided list and output them in order of most important to least important missing KC. This approach is a variation on previous KC extraction work which would ask the LLM to *generate* missing KCs from scratch rather than *classify* missing KCs from a provided list. We found that GPT-4 performed better if given the list of possible KCs when compared to expert labels. In a sample of 63 such queries, 85% of the LLM-generated missing KC lists were considered reasonable by experts (authors on this paper and also Teaching Assistants [TAs] for 61A). The two experts agreed on 93% of a small mutually labeled dataset.

Once we have the list of missing KCs associated with each code checkpoint, we look at time-adjacent pairs of checkpoints and compare whether the number of missing KCs decreases, increases, or stays the same. With this categorization, we can visualize how students' progressions through assignments changes with the introduction of 61A Bot as shown in Figure 5.1. Specifically, we calculate the average *time spent* per student and average *number of submissions* made per student. In our plots, green regions correspond to checkpoint pairs where the number of missing KCs decreases, red regions correspond to pairs where the number of missing KCs increases, and yellow regions correspond to pairs where the number of missing KCs stays the same.

From Spring 2023 to Spring 2024, we observe drops across the board both in time spent and number of submissions associated for each color region, which aligns with our completion time findings in § 4.4. Spending less time in yellow and red regions can be attributed to 61A Bot's hints pointing out issues in their code more quickly than the students themselves can. The reduction in the green region is less intuitive but can be explained by Spring 2024 students making *more progress per green submission* compared to Spring 2023 students. In Spring 2024, each green submission reduces the number of missing KCs by 1.42 on average (with a sample standard deviation of 0.71) while in Spring 2023, each green submission reduces the number of missing KCs by 1.28 on average (with a sample standard deviation of 0.56). Running a t-test here yields $p < 0.0000001$.

If we look at the time spent in each color category *proportionally* (see Figure 5.2), specifically in terms of the number of submissions (rather than time), we can see that the distribution of categories is mostly consistent from Spring 2023 to Spring 2024 (with some cases — e.g. HW3 — showing significant increases in green regions). This suggests that students' general progression through problems remains similar and is merely expedited by the bot in Spring 2024 i.e. students inevitably encounter and still struggle through resolving the same bugs and issues regardless of hints but overall move at a faster pace when the bot is available. See Appendix C.4 for plots of additional homeworks.

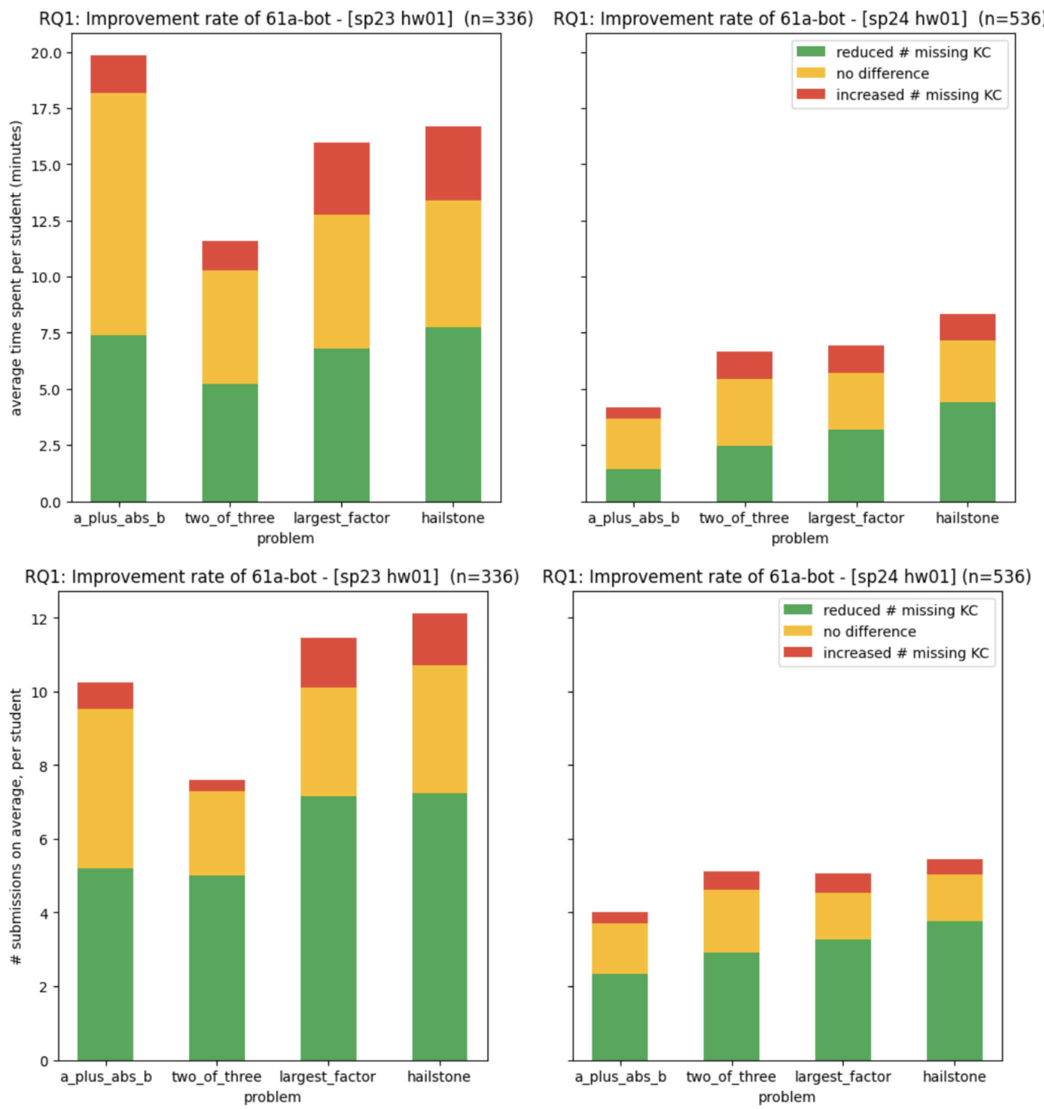

Figure 5.1: Sp23 (Left) vs Sp24 (Right) HW1 Time Spent & Number of Submissions Colored by Student Progression. Upper plots show average time spent per student. Lower plots show average number of submissions per student.

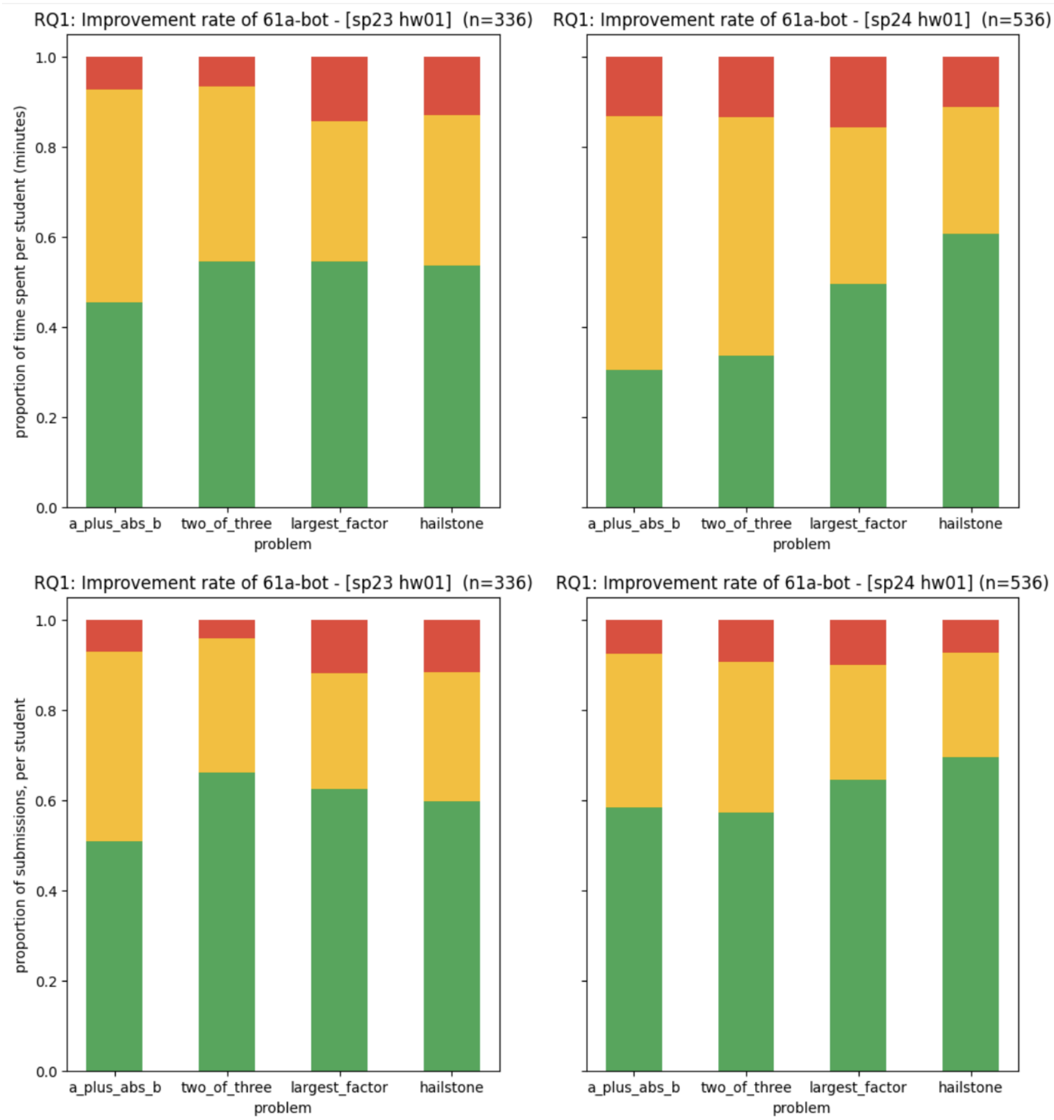

Figure 5.2: Sp23 (Left) vs Sp24 (Right) HW1 *Proportion* of Time Spent & Number of Submissions Colored by Student Progression. Upper plots show average time spent per student. Lower plots show average number of submissions per student.

5.2 RQ2: How effectively do the hints capture issues in the code?

Next, we explore which KCs the *hints* address and look at the overlap between the hint KCs and code KCs as a measure for how effectively 61A Bot hints are able to capture issues in student code. Following a similar process as in § 5.1, we pass the hint, a list of human-generated KCs, and some metadata (see Appendix A.3) to GPT-4 and ask it to determine which KCs the hint addresses from the provided list and, this time, output them in any order. In a sample of 62 similar queries, 87% of the time, the LLM correctly classified whether a KC existed in the hint when compared with expert labels (labeled by authors on this paper who are also TAs for 61A). The two experts agreed on 92% of a small mutually labeled dataset.

With the list of KCs addressed by each hint along with the list of missing KCs for each student code checkpoint from § 5.1, we can measure the overlap. Specifically, we calculate the proportion of hints that address *at least 1 out of the top 3* missing KCs in the student's code. We use this metric of "overlap with top 3 missing KCs" because our goal with 61A Bot is not to have its hints point out *every* single bug and misunderstanding the student has — instead, we are interested in whether the bot's hints are addressing the *most pressing* missing KCs in the student code, similar to how a human tutor would walk through a problem with a student in office hours. As seen in Figure 5.3, we notice that over 85% of hints from homework 1 fulfill this overlap criterion for *all* hint KC list lengths (i.e. hint KC-richness). This is surprising as we would expect hints that cover a larger number of KCs to have a higher likelihood of also covering one of the top 3 missing KCs in the student's code, but the success proportion appears nearly uniform over all hint KC list lengths. Another implication of this hint KC-richness independence is that the LLM is actually quite good at recognizing and weighing the importance of various issues in student code and ensuring that the *key* issues are included in its response — possibly an unintended but welcome side-effect of including the `Limit your response to a sentence or two at most` instruction in the prompt.

For later homeworks, we don't observe as strong an overlap between KCs addressed by hint and missing KCs in the code, specifically with hints that address fewer KCs (i.e. shorter hint KC list length). We speculate this is due to our few-shot expert examples being limited to samples from homework 1 leading to less accurate KC classifications both in the hints and the code checkpoints. The effect of this lower accuracy is exacerbated in the low-data settings of hints that only address one or two KCs. See Appendix C.5 for plots of additional homeworks.

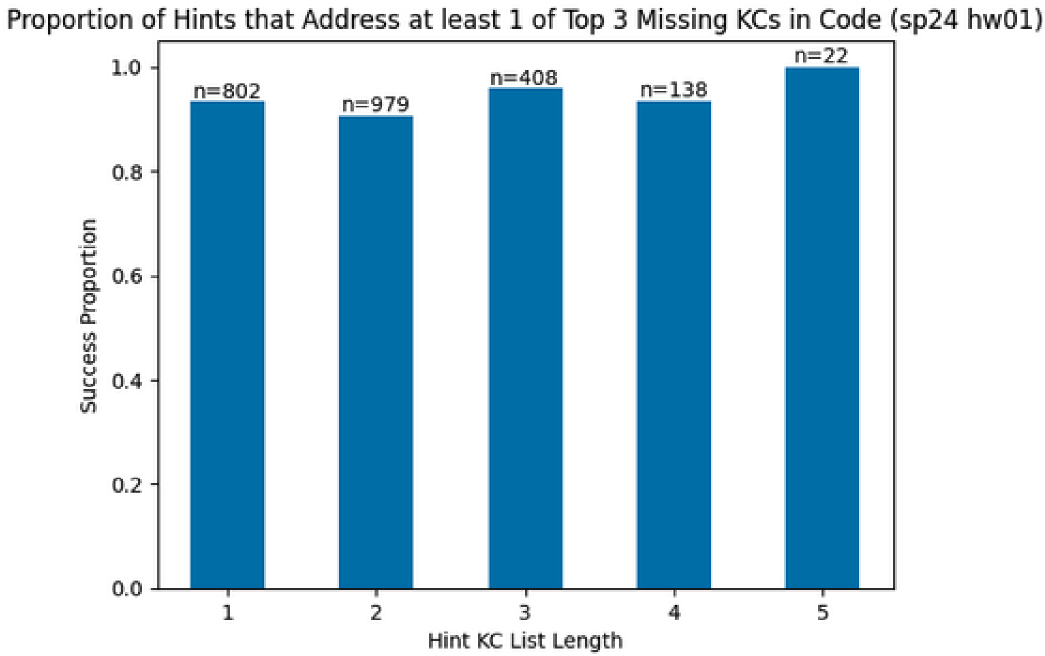

Figure 5.3: Sp24 HW1 Proportion of Hints that Address at least 1 of Top 3 Missing KCs in Code Grouped By Hint KC List Length

5.3 RQ3: Are students able to apply the hints to make progress?

Finally, to measure how well students are able to apply the hints to improve their code, we can start by using the same calculation as done before in § 5.2. Then, we introduce the same coloring scheme as in § 5.1 to indicate what proportion of hints that address a top 3 KC *are also* successfully applied by students to make progress. As we see in Figure 5.4 (and Appendix C.6), hints that cover a greater number of KCs are more likely to lead to direct student progress (green zones) and less likely to lead to minimal progress submissions (yellow zones). This suggests that steering the LLM to cover 3 or 4 issues in a single hint could lead to student progress more often than honing in on one bug — a bug which in reality could be a minor issue when compared to the student's general approach to the problem, for example. In contrast to the previous result in § 5.2, we see the number of KCs addressed by the hint playing a key role in students' ability to apply at least *one* of the suggestions in the hint. While KC-sparse hints may be able to address core issues in student code consistently, there's a better chance of the student successfully applying *some* part of the hint to make progress when the hint is KC-rich. See Appendix E.1 & E.2 for samples of green & yellow submission pairs respectively.

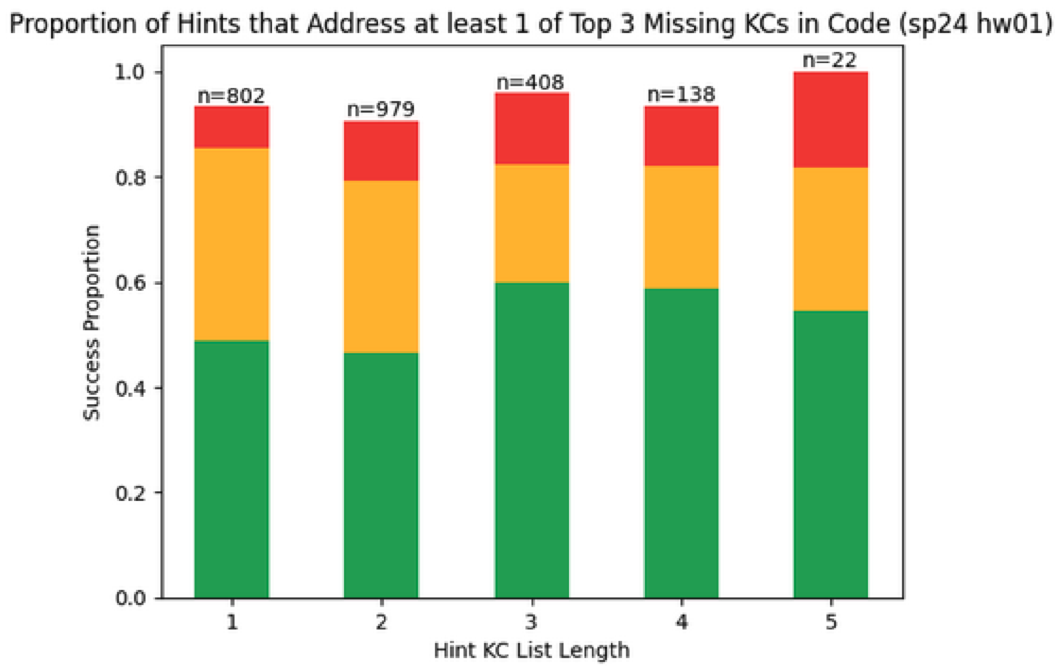

Figure 5.4: Sp24 HW1 Proportion of Hints that Address at least 1 of Top 3 Missing KCs in Code Colored by Student Progression and Grouped By Hint KC List Length

Interestingly, it seems that while the proportion of green increases and yellow decreases with more KC-rich hints, the proportion of red hints remains relatively consistent. One explanation for this is that there is an ever-present chance that students misunderstand what the hint wants them to do. In this scenario, which is independent of hint KC list length, it's likely that the student will make reverse progress. See Appendix E.3 for examples of red submission pairs.

Chapter 6

Next Steps

In the short term, we hope to continue reducing the error rate through additional tuning and training. To further augment the context, we intend to offer the student options to request different kinds of help and rate the helpfulness of the hint. Simple responses like "Please elaborate." or "Can you explain it another way?" could provide valuable context to steer subsequent prompts.

In the long term, we plan to leverage feedback to make larger-scale adjustments to the system as a whole. From student feedback in § 4.3 and KC analysis in § 5, it's evident that the current helpfulness of hints is good but has room for improvement. One avenue for achieving improvement is through human feedback. We are already collecting helpfulness rating feedback from students after each hint is provided which could be incorporated via reinforcement learning from human feedback (RLHF). First, we could use hints that received positive feedback from students to curate a dataset for fine-tuning. In fact, we have already experimented with this idea in an effort to fine-tune a model that was specifically tailored toward helping 61A students with Scheme homework problems. While it showed promise in its capability to dramatically improve GPT-3.5's hints, the fine-tuned GPT-3.5 model was still outperformed by base GPT-4. So, until fine-tuning is made available for SOTA models (or we decide to train and host our own open-source model, which comes with its own heap of challenges), we don't plan on implementing it.

Another avenue for improving hint quality is by training a completion time model. Recent work [14] has shown the potential for neural networks to classify student code as correct or incorrect by embedding code abstract syntax trees (ASTs). We can extend this to the regression task of estimating how long a student will take to complete an problem from their current code state. Furthermore, we can add the hints that 61A Bot previously gave in response to each code checkpoint as an additional input. With a $(code\_ast, hint) \mapsto time\_until\_completion$ model, we can treat remaining completion time as our objective to minimize and generate a fine-tuning dataset this way.

To address the issue of hints being too vague or too specific for individual students in § 4.3, we aim to make each hint more personalized. Using a mix of student checkpoint data, student feedback, and KCs we can generate a student model [30] for each student that summarizes their learning habits and the granularity of hints they prefer to provide as additional steering in the prompt. The primary concern here is latency, but the methodology is promising.

To extend the work we did in § 5, we plan to perform additional *time-series* analysis of student submissions to gain an extra dimension of insight into how individual students are progressing through assignments differently with and without access to the bot. Moreover, one clear drawback of the KC-based progression categorization we've developed is that the *number* of missing KCs is not always an accurate measure how much progress a student is making. For example, a code submission with a slew of "Off-by-One" and "Forget While Loop Update" errors could realistically be much closer to a correct solution than a recursive function that has a single "Incorrect Recursive Recombination Step" missing KC associated with it. So, collecting more data on this approach's validity (or shortcomings) will prove valuable.

Chapter 7

Conclusion

We first reported here on the development and deployment process of an LLM-based interactive programming support tool for introductory CS courses. We showed high-level effects of introducing 61A Bot including reduction in student's assignment completion times and reduction in student reliance on staff support. Additionally, we explored **knowledge components** as a proxy for student progress and found that students follow the same general progression trajectories through problems — only faster with 61A Bot. Analyzing the hints themselves revealed that the LLM *is* able to decipher important issues in student code at a high rate and that hints that cover a number of rather than a single bug are more likely to lead to direct student progress.

We discovered a number of successes, identified a few challenges and pitfalls, and reflected on solutions and paths toward a more pedagogically robust LLM assistant for introductory CS students.

Bibliography

[1] Brett A Becker et al. "Programming Is Hard–Or at Least It Used to Be: Educational Opportunities And Challenges of AI Code Generation". In: (2023).

[2] Mark Chen et al. "Evaluating large language models trained on code". In: *arXiv preprint arXiv:2107.03374* (2021).

[3] Bruno Pereira Cipriano and Pedro Alves. "GPT-3 vs Object Oriented Programming Assignments: An Experience Report". In: *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. 2023, pp. 61–67.

[4] Tyne Crow, Andrew Luxton-Reilly, and Burkhard Wuensche. "Intelligent tutoring systems for programming education: a systematic review". In: *Proceedings of the 20th Australasian Computing Education Conference*. 2018, pp. 53–62.

[5] Franklin Demana and BK Waits. "Calculators in mathematics teaching and learning". In: *Past, present, and future. In Learning Mathematics for a New Century* (2000), pp. 51–66.

[6] Paul Denny et al. "Chat Overflow: Artificially Intelligent Models for Computing Education-renAIssance or apocAIypse?" In: *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. 2023, pp. 3–4.

[7] Paul Denny et al. "Computing Education in the Era of Generative AI". In: *arXiv preprint arXiv:2306.02608* (2023).

[8] Paul Denny et al. "Promptly: Using Prompt Problems to Teach Learners How to Effectively Utilize AI Code Generators". In: *arXiv preprint arXiv:2307.16364* (2023).

[9] Katherine Donlevy. *Harvard to roll out AI professors in flagship coding class for fall semester.* 2023. URL: https://nypost.com/2023/06/30/harvard-to-roll-out-ai-professors-in-flagship-coding-class-for-fall-semester/.

[10] Ed. 2024. URL: https://edstem.org/.

[11] James Finnie-Ansley et al. "My AI Wants to Know if This Will Be on the Exam: Testing OpenAI's Codex on CS2 Programming Exercises". In: *Proceedings of the 25th Australasian Computing Education Conference*. 2023, pp. 97–104.

[12] James Finnie-Ansley et al. "The robots are coming: Exploring the implications of openai codex on introductory programming". In: *Proceedings of the 24th Australasian Computing Education Conference*. 2022, pp. 10–19.

[13] Arto Hellas et al. "Exploring the Responses of Large Language Models to Beginner Programmers' Help Requests". In: *Proceedings of the 2023 ACM Conference on International Computing Education Research V.1*. 2023.

[14] Muntasir Hoq et al. "SANN: Programming Code Representation Using Attention Neural Network with Optimized Subtree Extraction". In: *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*. CIKM '23. Birmingham, United Kingdom: Association for Computing Machinery, 2023, pp. 783–792. ISBN: 9798400701245. DOI: 10.1145/3583780.3615047. URL: https://doi.org/10.1145/3583780.3615047.

[15] Majeed Kazemitabaar et al. "Studying the effect of AI Code Generators on Supporting Novice Learners in Introductory Programming". In: *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 2023, pp. 1–23.

[16] K. Koedinger, Albert T. Corbett, and Charles A. Perfetti. "The Knowledge-Learning-Instruction (KLI) Framework: Toward Bridging the Science-Practice Chasm to Enhance Robust Student Learning". In: 2010. URL: https://api.semanticscholar.org/CorpusID:15732814.

[17] Juho Leinonen et al. "Comparing Code Explanations Created by Students and Large Language Models". In: *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. ACM, 2023, pp. 124–130. ISBN: 9798400701382.

[18] Juho Leinonen et al. "Using Large Language Models to Enhance Programming Error Messages". In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. ACM. 2023.

[19] Yujia Li et al. "Competition-level code generation with alphacode". In: *Science* 378.6624 (2022), pp. 1092–1097.

[20] Rongxin Liu et al. "Teaching CS50 with AI: Leveraging Generative Artificial Intelligence in Computer Science Education". In: *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2*. SIGCSE 2024. Portland, OR, USA: Association for Computing Machinery, 2024, p. 1927. ISBN: 9798400704246. DOI: 10.1145/3626253.3635427. URL: https://doi.org/10.1145/3626253.3635427.

[21] Stephen MacNeil et al. "Experiences from using code explanations generated by large language models in a web software development e-book". In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 2023, pp. 931–937.

[22] Stephen MacNeil et al. "The Implications of Large Language Models for CS Teachers and Students". In: *Proc. of the 54th ACM Technical Symposium on Computer Science Education*. Vol. 2. 2023.

[23] Microsoft. *Azure OpenAI*. 2024. URL: https://azure.microsoft.com/en-us/products/ai-services/openai-service/.

[24] Samim Mirhosseini, Austin Z Henley, and Chris Parnin. "What is your biggest pain point? an investigation of cs instructor obstacles, workarounds, and desires". In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 2023, pp. 291–297.

[25] Chancharik Mitra et al. "Elevating Learning Experiences: Leveraging Large Language Models as Student-Facing Assistants in Discussion Forums". In: *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2*. SIGCSE 2024. Portland, OR, USA: Association for Computing Machinery, 2024, pp. 1752–1753. ISBN: 9798400704246. DOI: 10.1145/3626253.3635609. URL: https://doi.org/10.1145/3626253.3635609.

[26] Rose Niousha et al. "Use of Large Language Models for Extracting Knowledge Components in CS1 Programming Exercises". In: *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 2*. SIGCSE 2024. Portland, OR, USA: Association for Computing Machinery, 2024, pp. 1762–1763. ISBN: 9798400704246. DOI: 10.1145/3626253.3635592. URL: https://doi.org/10.1145/3626253.3635592.

[27] Erik Ofgang. "What is Khanmigo? The GPT-4 learning tool explained by Sal Khan". In: *Tech & Learn* (2023).

[28] OpenAI. *GPT-4*. 2023. URL: https://openai.com/index/gpt-4-research/.

[29] Zachary A Pardos and Shreya Bhandari. "Learning gain differences between ChatGPT and human tutor generated algebra hints". In: *arXiv preprint arXiv:2302.06871* (2023).

[30] Chris Piech et al. "Modeling how students learn to program". In: *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*. SIGCSE '12. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 153–160. ISBN: 9781450310987. DOI: 10.1145/2157136.2157182. URL: https://doi.org/10.1145/2157136.2157182.

[31] James Prather et al. "" It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers". In: *arXiv preprint arXiv:2304.02491* (2023).

[32] James Prather et al. "Transformed by transformers: Navigating the ai coding revolution for computing education: An iticse working group conducted by humans". In: *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 2*. 2023, pp. 561–562.

[33] Brent Reeves et al. "Evaluating the Performance of Code Generation Models for Solving Parsons Problems With Small Prompt Variations". In: *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. 2023, pp. 299–305.

[34] Sami Sarsa et al. "Automatic generation of programming exercises and code explanations using large language models". In: *Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*. 2022, pp. 27–43.

[35] Ryo Suzuki et al. "Tracediff: Debugging unexpected code behavior using trace divergences". In: *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2017, pp. 107–115.

[36] Sierra Wang, John Mitchell, and Chris Piech. "A Large Scale RCT on Effective Error Messages in CS1". In: *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2024. Portland, OR, USA: Association for Computing Machinery, 2024, pp. 1395–1401. ISBN: 9798400704239. DOI: 10.1145/3626252.3630764. URL: https://doi.org/10.1145/3626252.3630764.

[37] Yu-Chieh Wu, Andrew Petersen, and Lisa Zhang. "Student Reactions to Bots on Course Q&A Platform". In: *Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 2*. 2022, pp. 621–621.

[38] Tongshuang Wu, Kenneth Koedinger, et al. "Is AI the better programming partner? Human-Human Pair Programming vs. Human-AI pAIr Programming". In: *arXiv preprint arXiv:2306.05153* (2023).

[39] Zhiqiang Yuan et al. "Evaluating instruction-tuned large language models on code comprehension and generation". In: *arXiv preprint arXiv:2308.01240* (2023).

[40] J.D. Zamfirescu-Pereira et al. *Conversational Programming with LLM-Powered Interactive Support in an Introductory Computer Science Course*. New Orleans, LA, USA, 2023. URL: https://gaied.org/neurips2023/files/32/32_paper.pdf.

[41] J.D. Zamfirescu-Pereira et al. "Herding AI Cats: Lessons from Designing a Chatbot by Prompting GPT-3". In: *Proceedings of the 2023 ACM Designing Interactive Systems Conference*. Pittsburgh, PA, USA, 2023, pp. 2206–2220. URL: https://doi.org/10.1145/3563657.3596138.

Appendix A

Prompts

A.1 61A Bot Prompt

Our final prompt is as follows. Note that the `%NOTE%` marker is replaced by text specific to the homework assignment selected by the student.

```
You are a very talented 61a tutoring bot, the intro CS class at UC Berkeley; you are helping students learn to program.

A student has asked for help. The question they are trying to solve is in the next message, and in the following message is (all) the code the student has so far, including some code that is for other questions -- ignore that other code. If the student asks for help repeatedly, the conversation will continue with your subsequent reply and any updated student code.

First: identify the student's code. Is it correct and complete? If so, tell them "That looks like it should work -- give it a shot and feel free to include any error message in a comment!"

If not, here's what you should analyze, based on their code:

1) Is the student missing any conceptual knowledge? Would a refresher help?

2) Is the code they have already, if any, on the right track?

3) How close are they to a working solution?

4) Did they follow your previous instructions? If not, rephrase and offer advice in a different way.
```

```
5) Do they have a plan -- if not, help them generate one.

6) If all else fails, provide a template of the code -- perhaps missing the key recursive case and base case if condition.

%NOTE%

Do not give the student the answer or any code. If there's an obvious bug, direct them to the location of the bug. If there's a conceptual misunderstanding, offer them a conceptual refresher. Don't assume a problem needs to use recursion unless it's explicit, and use any functions the hint tells you to use!

Limit your response to a sentence or two at most. Be as socratic as possible, and be super friendly.

Handing you off to the student now!
```

A.2 RQ1 Prompt

```
You are a smart teacher and a classifier for an introductory computer science course.
Your job is to classify student's missing knowledge component(s) based on the code submission they gave.
Each input for classification will have the following parts:

<Submission>: includes the student's code for the question
<Doctest Percentage>: indicates the percentage of doctests the student passed, ranging from [0.0, 1.0] OR 'Infinite Loop' if the student's code runs for too long.
<Current Question>: indicates which question/function the student is currently working on. Any code in the <Submission> for other functions should be ignored.
<Skeleton>: includes the original skeleton code for the current question. Students should not modify code provided in the skeleton and should instead mold their solution around the provided skeleton code.



Classification:
Based on the student submission log and the correct answer, specify and list out the knowledge component that the student might be missing.
For example, if there is an answer that is missing a necessary return statement, that would be listed as "Forget Return"

Reasoning:
Also provide a reason why that classification was made.

Output:
As there can be multiple components, output a list of lists that contains all the missing knowledge components. The first element contains the list of missing knowledge components.
The second element contains the reason behind that list of missing knowledge components identified.


Example Output: ["Arithmetic Logic", "Forget Return"], The reason why it was listed as these two components is because...]"

```

```
The possible list of missing knowledge components is {KC_LIST}. Pick the relevant missing knowledge components from the provided list only.
Here are descriptions of each knowledge component:
{KC_DESC_STR}

If the student passed all doctests (i.e. doctest percentage is 1.0), you can write 'Correct'. If the student's submission is completely blank (i.e. an EXACT copy of the skeleton code), you can write 'Blank'.

Rank the list of knowledge components in the order of importance toward arriving at a solution. In other words, the most pressing missing knowledge component comes first.

Make sure to wrap each element with same quotation marks - ". Do not use '
In other words, it can be directly converted to a list of list.
If I index the output[0], this should give me the list of knowledge components
If I index the output[1], this should give me the reason in text

The first element is a list. The second element is a string.
These two elements have to be wrapped with an outer list.
So once again, output format is:

"[knowledge-component1, knowledge-component2, ...], reason"
ex) ["Assignment Statement", "Forget Return"], '"The student seems to have wrong variable assignments. In addition, ..."
We want to strictly follow this format, as I want to further apply list comprehension and use the output as a list of lists. Make sure to follow the format described.

Finally, here are a few example submissions and expert classifications that you should use as a guide for generating your own responses:
{HAILSTONE_EXAMPLES_STRING}
{A_PLUS_ABS_B_EXAMPLES_STRING}

Now, here comes the submission you need to classify!
```

A.3 RQ2 Prompt

```
You are a smart teacher and classifier for an intro computer science course in Python.
In this course, students submit code and are given hints in response to address issues in their code. Your job is to classify which knowledge component(s) the hint targets.

Each input for classification will have the following parts:

<Hint>: the hint that was given in response to the student's code.
<Submission>: the student's code.

As well as some METADATA:
<Current Question>: which question/function the student is currently working on. Any code in the <Submission> for other functions should be ignored.
<Skeleton>: the original skeleton code for the current question. Students should not modify code provided in the skeleton and should instead mold their solution around the provided skeleton code.
<Doctest Percentage>: the percentage of doctests the student passed, ranging from [0.0, 1.0] OR 'Infinite Loop' if the student's code runs for too long.

Classification:
List the knowledge component(s) that the hint addresses.
For example, if the hint tells the student they should add the line `i += 1` at the end of their while loop, the list should include "Forget While Loop Update" as one of the knowledge components that was addressed. There can be multiple knowledge components addressed in the hint.

Here is the list of possible knowledge components:
{KC_LIST}
Pick the relevant knowledge components from the above provided list only.

Here are descriptions of each knowledge component:
{KC_DESC_STR}

Reasoning:
Finally, output a string that summarizes your reasoning for each of the knowledge components you identified in the hint.
```

```
If the hint tells the student their code is completely correct, you should write 'Correct'. If the hint says the student hasn't started the problem yet (i.e. the student submits an EXACT copy of the skeleton code), you should write 'Blank'.

Output:
Start with a scratch space in which you think out loud through your reasoning and decide which knowledge components are relevant.
The format of your output is as follows:
- The first element contains the list of knowledge components in any order.
- The second element contains the reason behind that list of knowledge components identified.

Example Output: ["Arithmetic Logic", "Forget Return"], "The hint targets \"Arithmetic Logic\" as it directly references the student's misunderstanding of the modulus operator..."

If I index the output[0], this should give me the list of knowledge components.
If I index the output[1], this should give me the reason as a string.
Make sure to wrap each element with the same quotation marks - ". Do not use '.

The first element is a list of strings. The second element is a string.
These two elements must be wrapped in an outer list.
So once again, here is the output format 3 equivalent forms/examples:

1. list[str], str

2. "[knowledge-component1, knowledge-component2, ...], reason"

3. ["Assignment Statement", "Forget Return"], "The hint directly tells the student to fix their variable assignments. In addition, ..."

We want to strictly follow this format, as your output will be read programmatically. Make sure to follow the format described.

Finally, here are a few example hints and expert classifications that you should use as a guide for generating your own responses. Note that these example inputs do not include the METADATA that you will receive with your input:

```

```
{RQ2_HINT_EXAMPLES_STRING}

Now, here comes the submission you need to classify!
```

Appendix B

Hint Samples

The following two samples are each a log of some student's submission subsequence (including the code submitted and hint given in response at each submission index) that exemplify a failure mode described in § 4.2.

Homework problem statements can be found at `https://inst.eecs.berkeley.edu/~cs61a/sp23/hw/hw01/`.

B.1 Incorrect Application of Hint

````
<STUDENT CODE @ 2024-01-24 14:55:20+00:00>:
```def hailstone(n):
    i = 1
    while n > 1:
        if n % 2 == 0:
            n = n // 2
            print(n)
        else:
            n = n * 3 + 1
            print(n)
        i += 1
        return i
    return 1
<HINT>: Great job on the `a_plus_abs_b` function! However, there's a small
   mistake in your `hailstone` function. The `return i` statement is inside
    the while loop, which means the function will exit after the first
   iteration. Try moving it outside the loop so that it returns the total
   count after the loop has finished.
----------
<STUDENT CODE @ 2024-01-24 14:59:09+00:00>:
````
````

`````
````def hailstone(n):
    i = 1
    if n > 1:
        if n % 2 == 0:
            n = n // 2
            print(n)
        else:
            n = n * 3 + 1
            print(n)
        i += 1
    return i
    if n == 1:
        return 1
<HINT>: You're almost there! However, your `if n > 1:` condition is not
   inside a loop, so it will only execute once. You need to put this inside
    a while loop so that it continues to execute until `n` is 1. Also,
   remember to print `n` at the start of each iteration, not just after it'
   s been modified.
----------
<STUDENT CODE @ 2024-01-24 15:04:36+00:00>:
```def hailstone(n):
    i = 1
    if n > 1:
        if n % 2 == 0:
            print(n)
            n = n // 2
        else:
            print(n)
            n = n * 3 + 1
        i += 1
    return i
    if n == 1:
        print(n)
        return 1
<HINT>: You're almost there! However, your `if n > 1:` condition is still
   not inside a loop, so it will only execute once. You need to put this
   inside a while loop so that it continues to execute until `n` is 1. Also
   , remember to print `n` at the start of each iteration, not just after
   it's been modified.
`````

B.2 No Application of Hint

````
<STUDENT CODE @ 2024-01-24 13:59:46+00:00>:
```from operator import add, sub
def a_plus_abs_b(a, b):
    if b < 0:
        f = sub(a, b)
    else:
        f = add(a, b)
    return f(a, b)
```
<HINT>: You're very close! However, in the line `f = sub(a, b)` and `f = add
   (a, b)`, you should assign `f` to the function `sub` or `add`, not the
   result of the function. Try removing `a` and `b` from these lines.
----------
<STUDENT CODE @ 2024-01-24 15:12:26+00:00>
```from operator import add, sub
def a_plus_abs_b(a, b):
    if b < 0:
        f = sub(a, b)
    else:
        f = add(a, b)

    return f(a, b)
```
<HINT>: Great job! Your `hailstone` function is now correctly structured
   with a while loop and prints `n` at the start of each iteration. However
   , there's a small mistake in your `a_plus_abs_b` function. You're
   calling `f(a, b)` where `f` is `sub(a, b)` or `add(a, b)`. This means
   you're actually calling `sub(a, b)(a, b)` or `add(a, b)(a, b)`, which is
    not correct. You should assign `f` to `sub` or `add`, not `sub(a, b)`
   or `add(a, b)`.
----------
<STUDENT CODE @ 2024-01-24 15:14:22+00:00>
```from operator import add, sub
def a_plus_abs_b(a, b):
    if b < 0:
        f = sub(a, b)
    else:
        f = add(a, b)
````
````

````
    return f(a, b)
```
<HINT> You're almost there! In your `a_plus_abs_b` function, you should
   assign `f` to `sub` or `add`, not `sub(a, b)` or `add(a, b)`. So, it
   should be `f = sub` when `b < 0` and `f = add` otherwise. Then, you can
   call `f(a, b)` in your return statement.
````
````

Appendix C

Additional Plots

C.1 Problem-Level CDFs

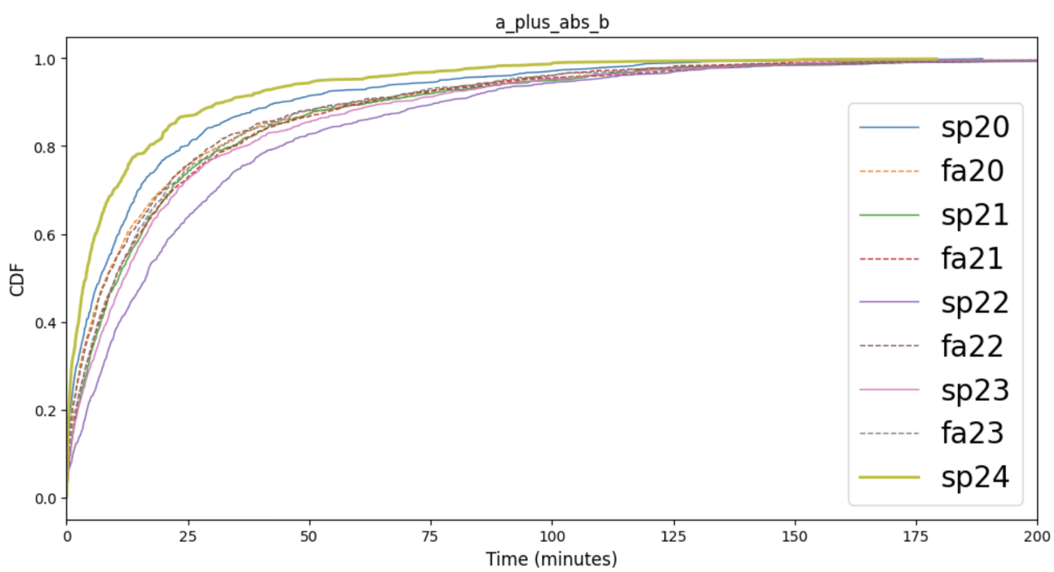

Figure C.1: `a_plus_abs_b` CDFs

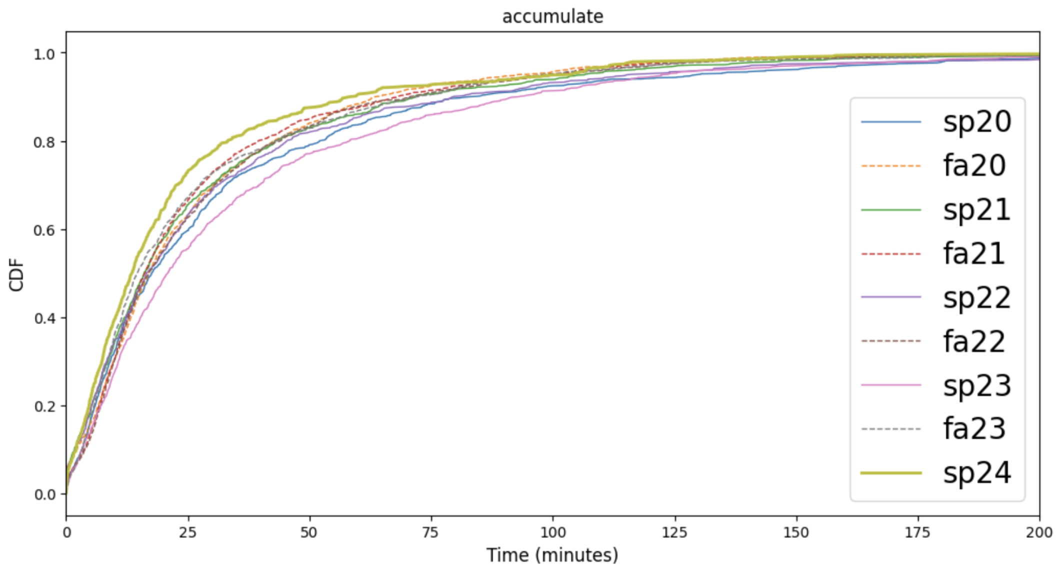

Figure C.2: `accumulate` CDFs

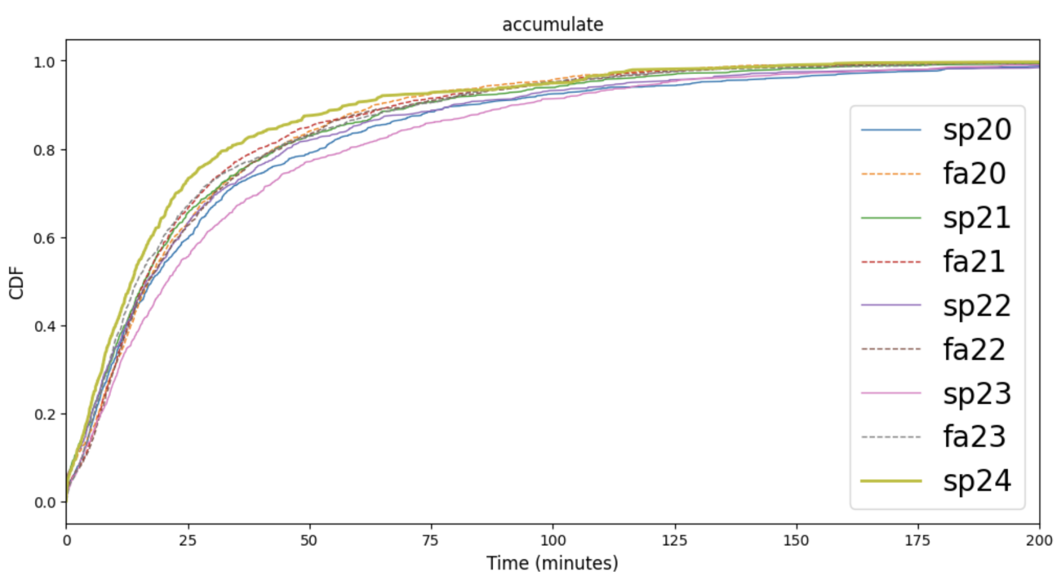

Figure C.3: `accumulate` CDFs

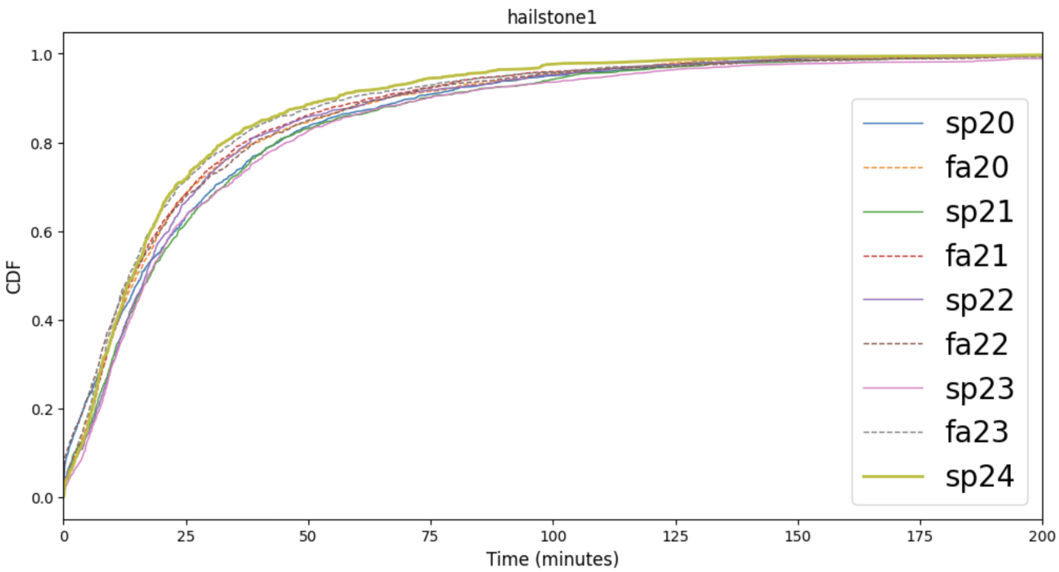

Figure C.4: `hailstone` CDFs

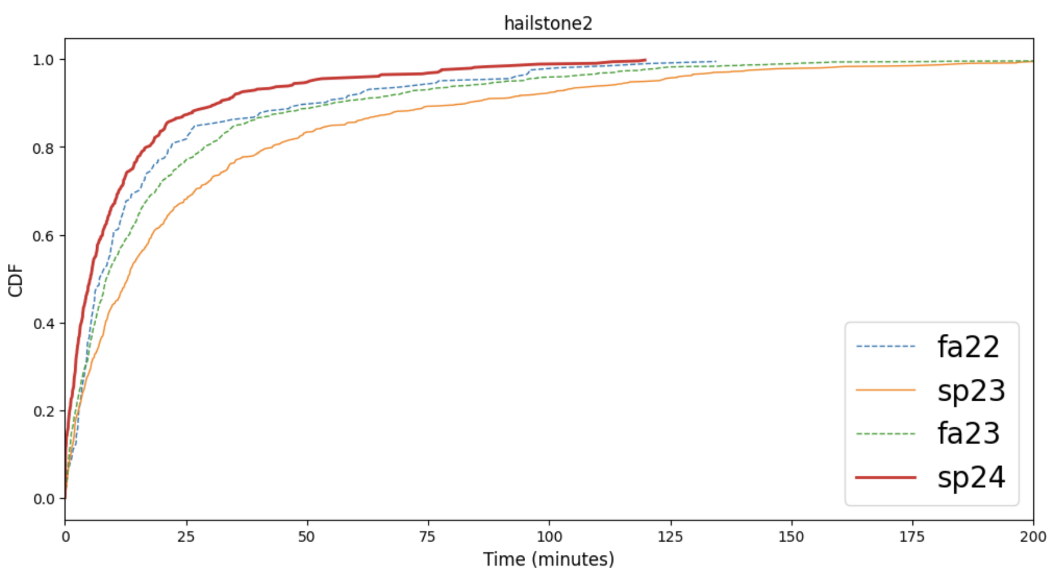

Figure C.5: `hailstone2` CDFs

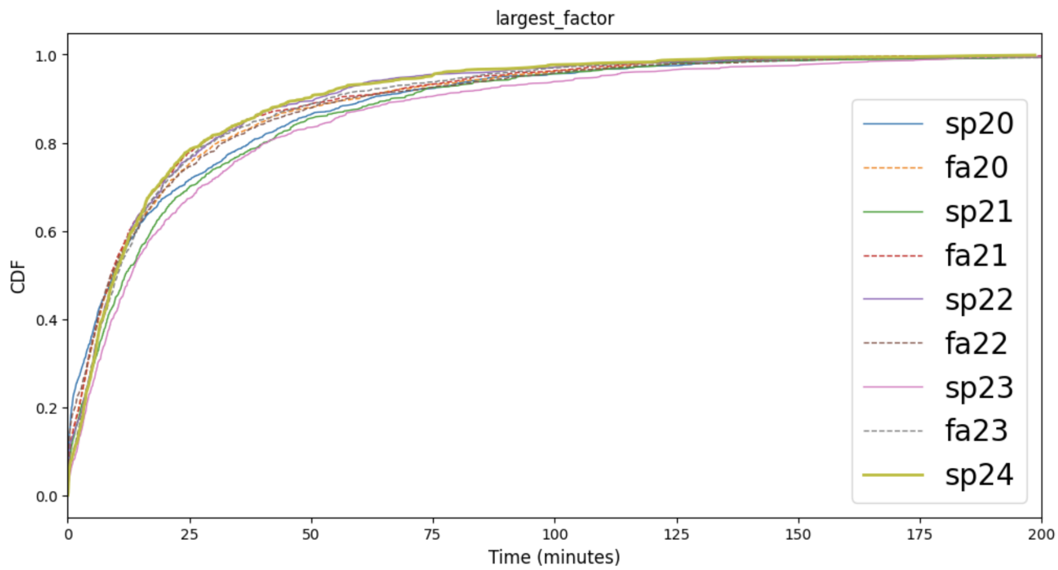

Figure C.6: `largest_factor` CDFs

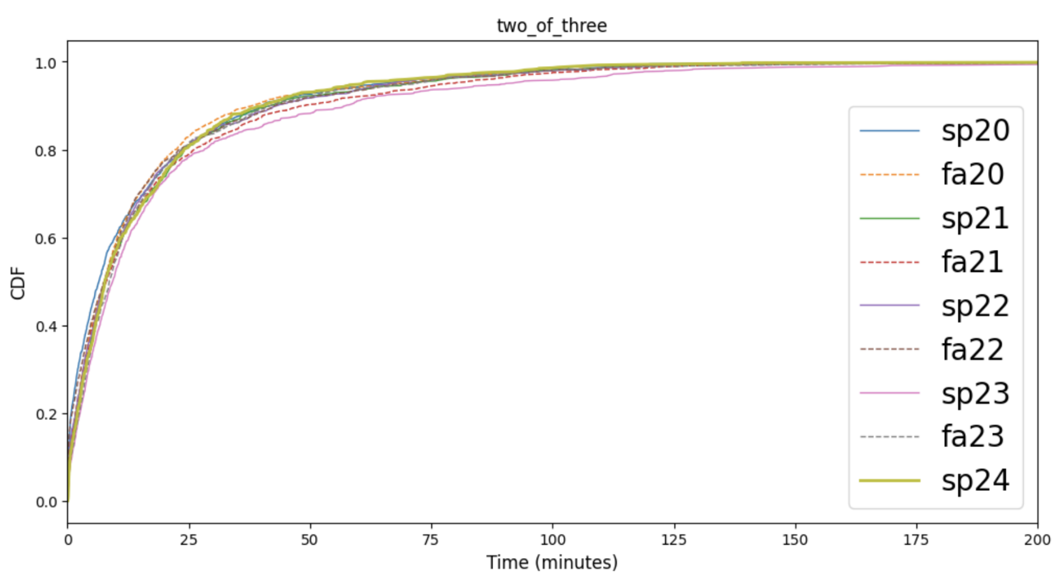

Figure C.7: `two_of_three` CDFs

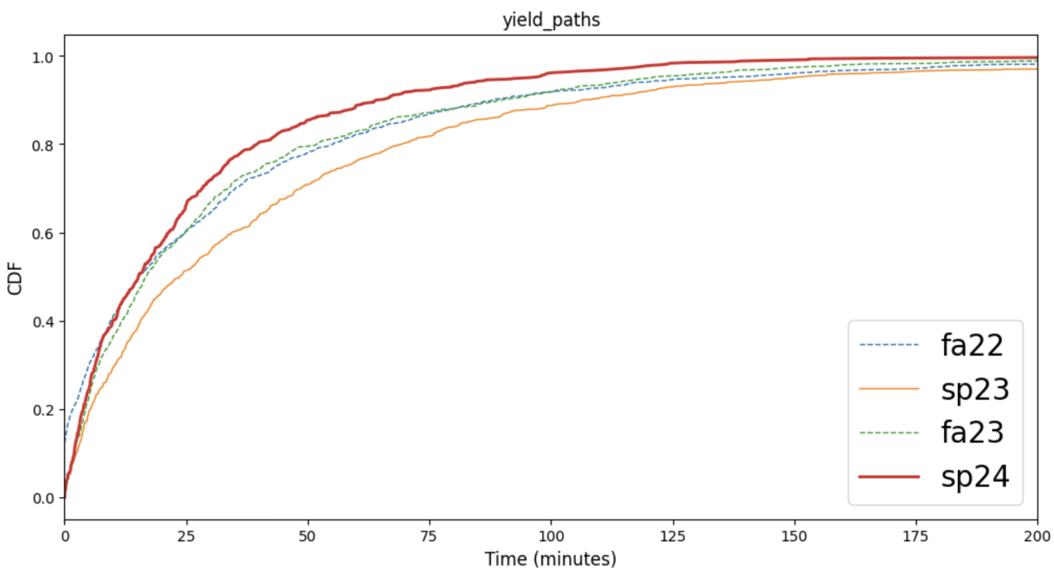

Figure C.8: `yield_paths` CDFs

C.2 Homework-Level CDFs

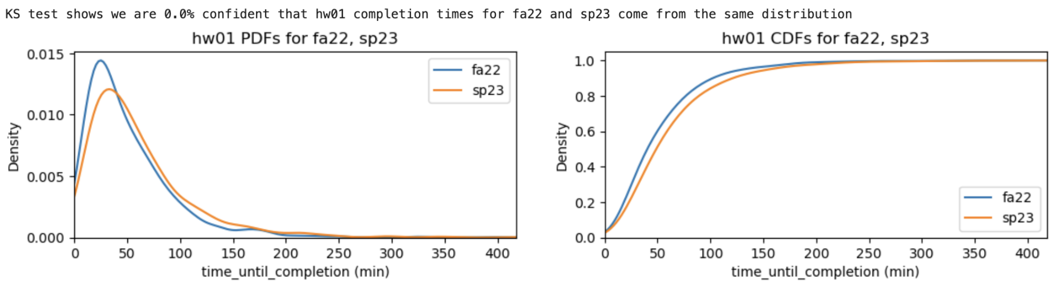

Figure C.9: Fall 2022 vs Spring 2023 HW 1 Completion Time Distributions (note: any density assigned to negative completion time is due to smoothing)

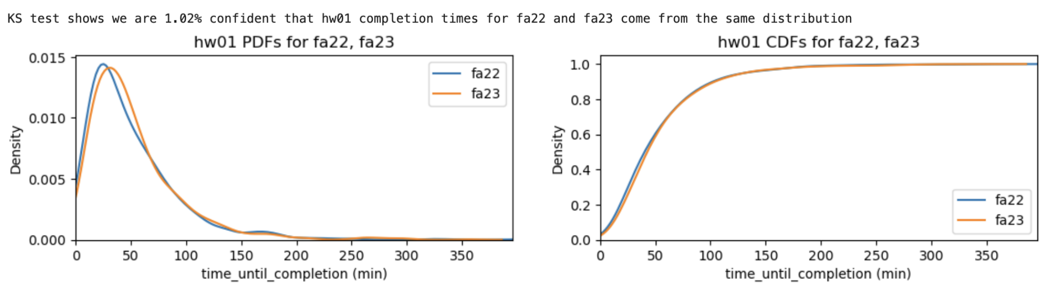

Figure C.10: Fall 2022 vs Fall 2023 HW 1 Completion Time Distributions (note: any density assigned to negative completion time is due to smoothing)

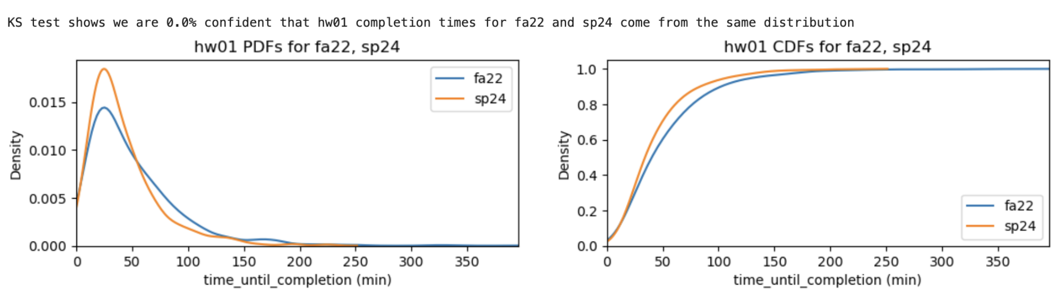

Figure C.11: Fall 2022 vs Spring 2024 HW 1 Completion Time Distributions (note: any density assigned to negative completion time is due to smoothing)

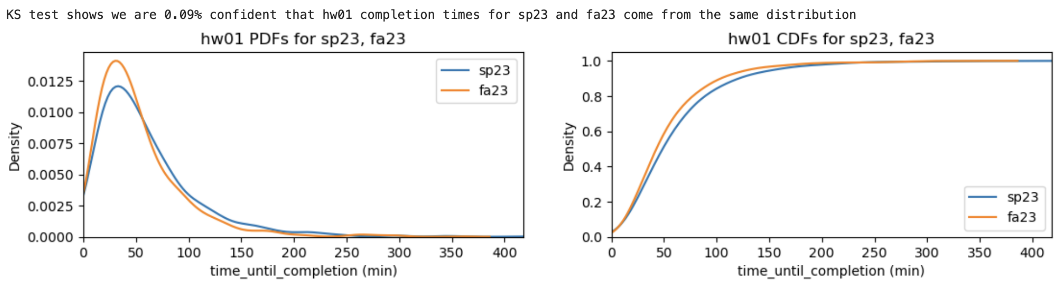

Figure C.12: Spring 2023 vs Fall 2023 HW 1 Completion Time Distributions (note: any density assigned to negative completion time is due to smoothing)

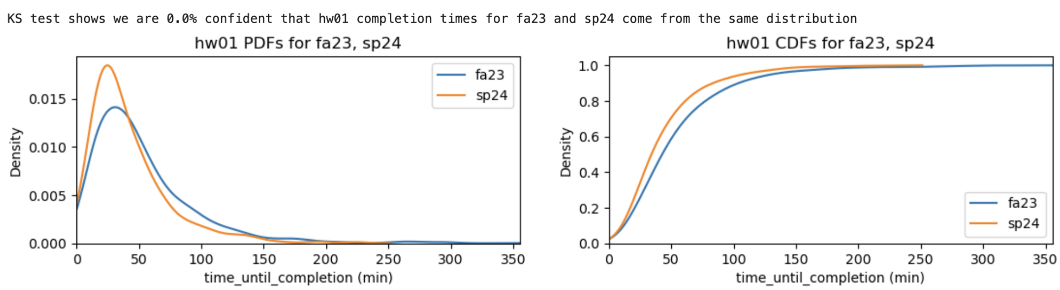

Figure C.13: Fall 2023 vs Spring 2024 HW 1 Completion Time Distributions (note: any density assigned to negative completion time is due to smoothing)

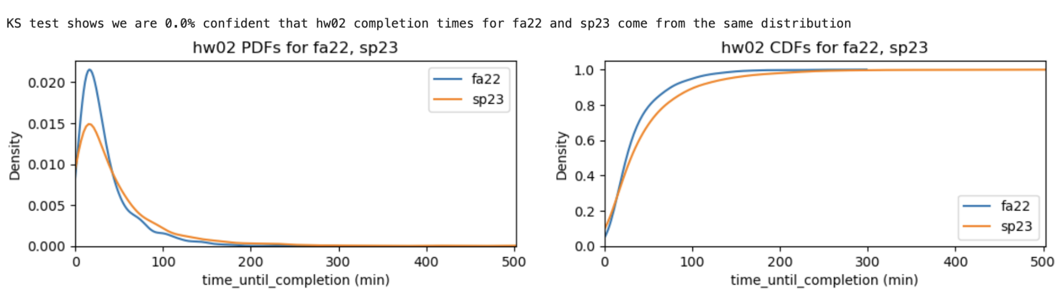

Figure C.14: Fall 2022 vs Spring 2023 HW 2 Completion Time Distributions (note: any density assigned to negative completion time is due to smoothing)

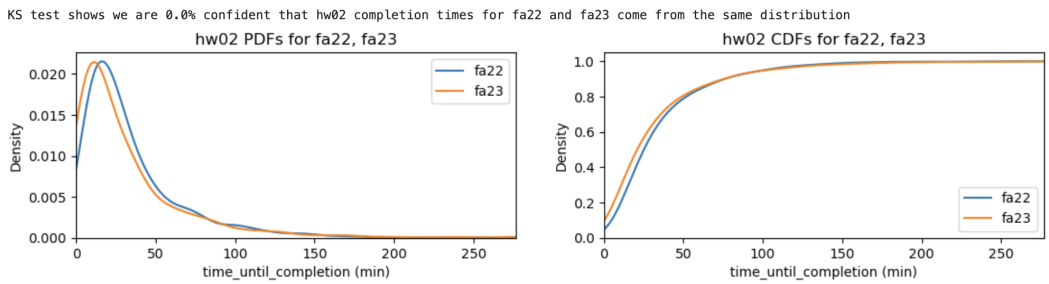

Figure C.15: Fall 2022 vs Fall 2023 HW 2 Completion Time Distributions (note: any density assigned to negative completion time is due to smoothing)

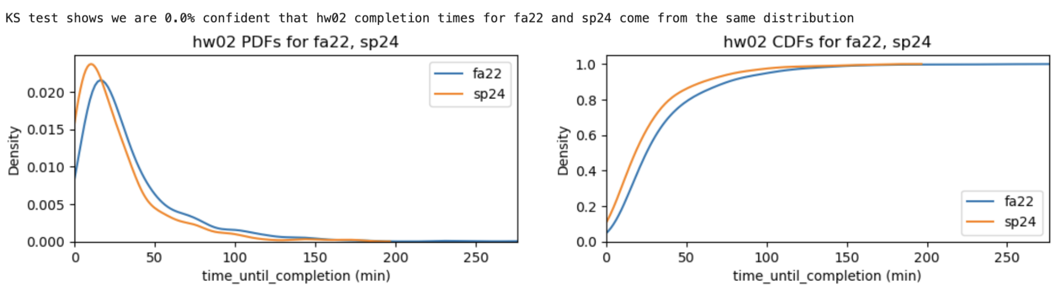

Figure C.16: Fall 2022 vs Spring 2024 HW 2 Completion Time Distributions (note: any density assigned to negative completion time is due to smoothing)

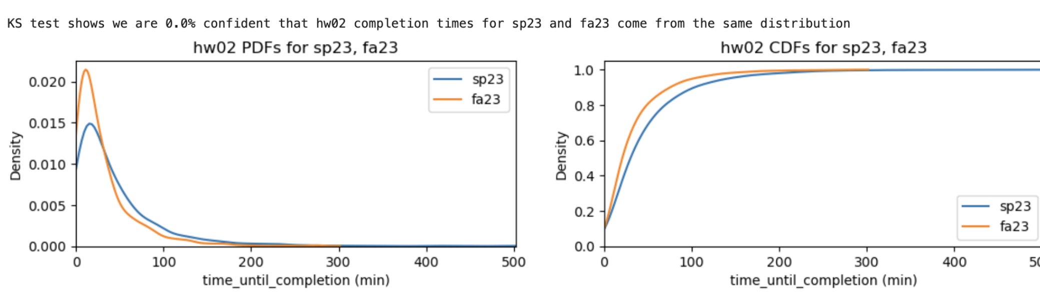

Figure C.17: Spring 2023 vs Fall 2023 HW 2 Completion Time Distributions (note: any density assigned to negative completion time is due to smoothing)

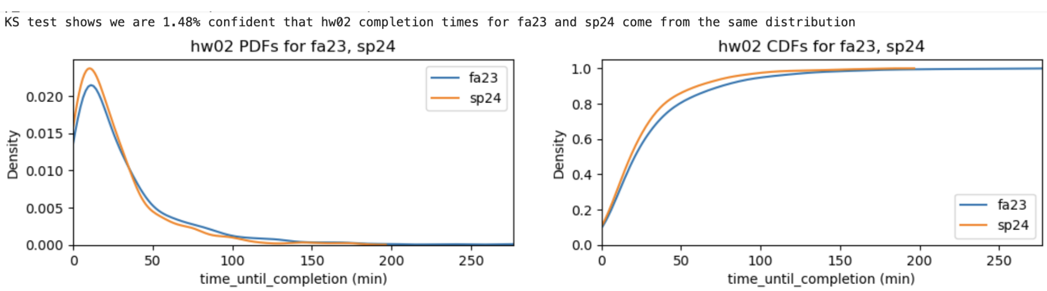

Figure C.18: Fall 2023 vs Spring 2024 HW 2 Completion Time Distributions (note: any density assigned to negative completion time is due to smoothing)

C.3 Ed Analytics

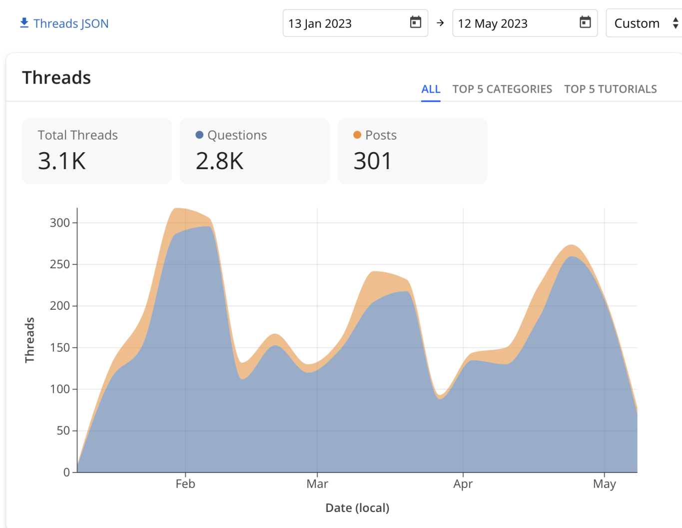

Figure C.19: Spring 2023 61A Ed Analytics.

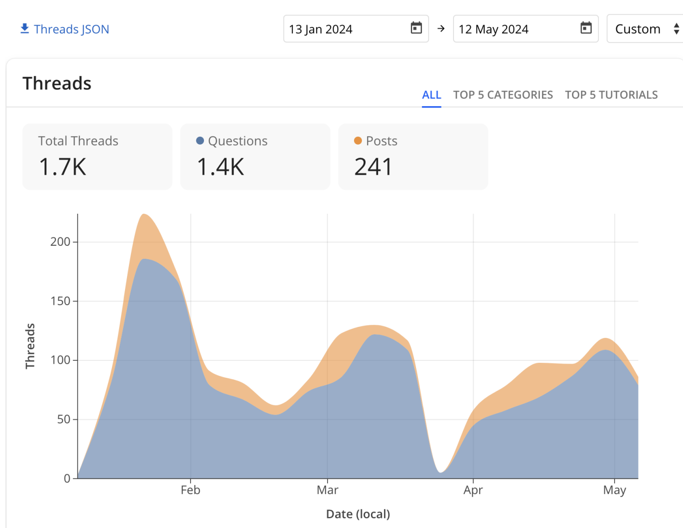

Figure C.20: Spring 2024 61A Ed Analytics.

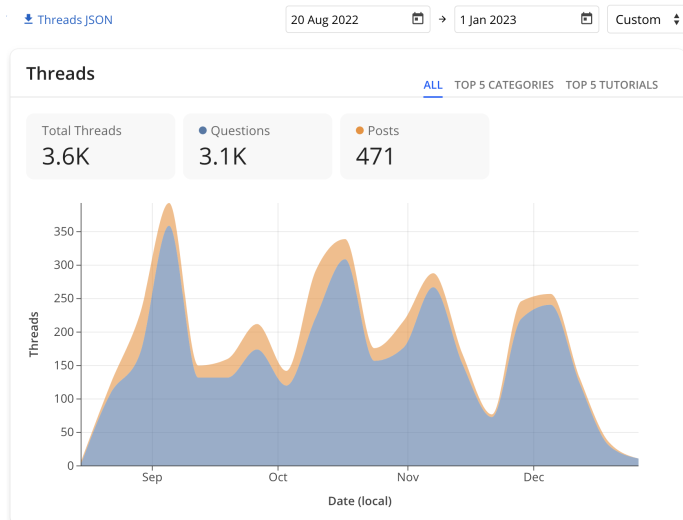

Figure C.21: Fall 2022 61A Ed Analytics.

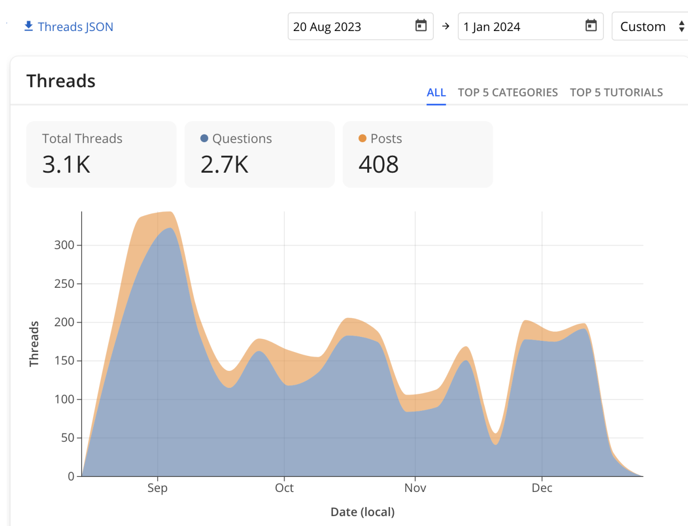

Figure C.22: Fall 2023 61A Ed Analytics.

C.4 RQ1 Plots

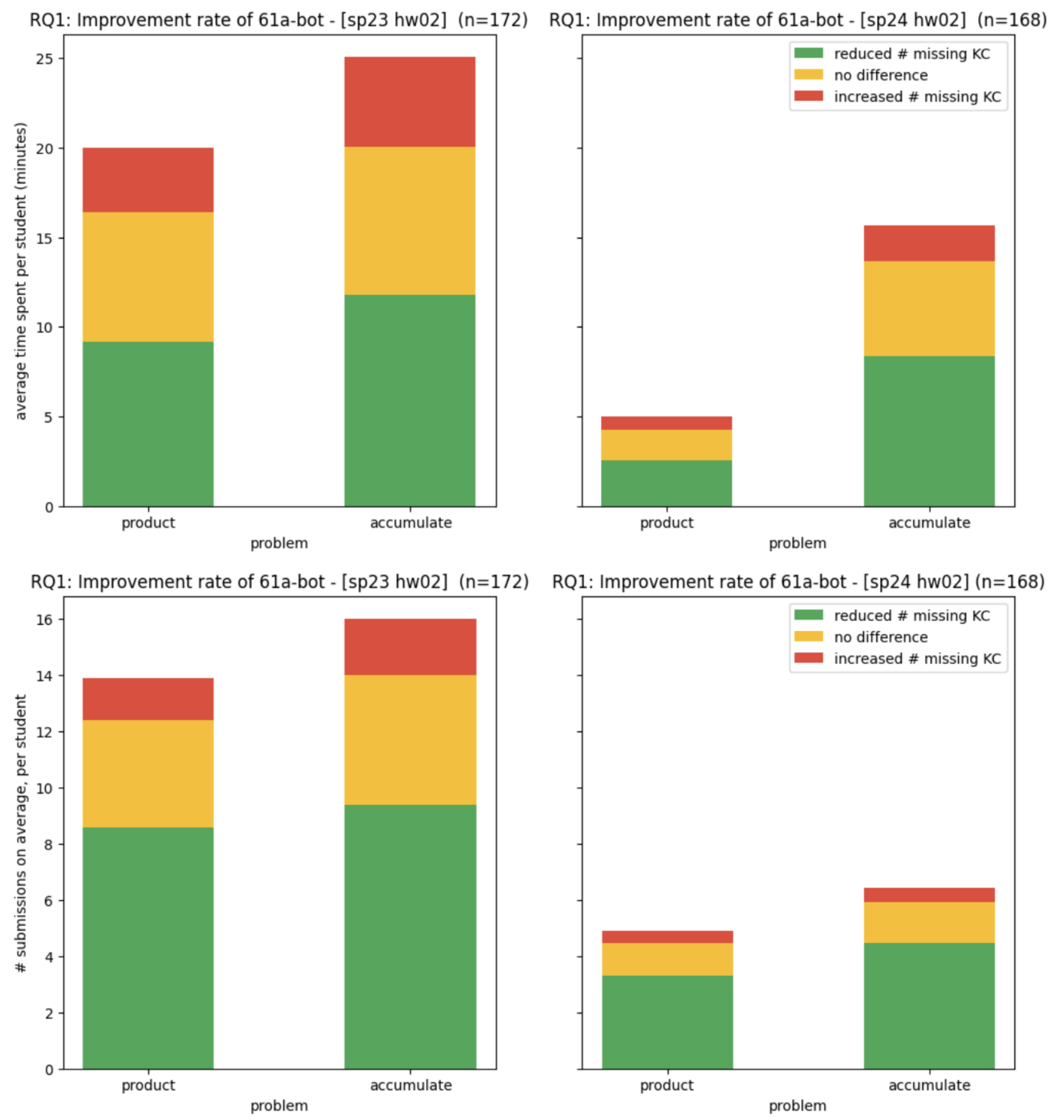

Figure C.23: Sp23 (Left) vs Sp24 (Right) HW2 Time Spent & Number of Submissions Colored by Student Progression. Upper plots show average time spent per student. Lower plots show average number of submissions per student.

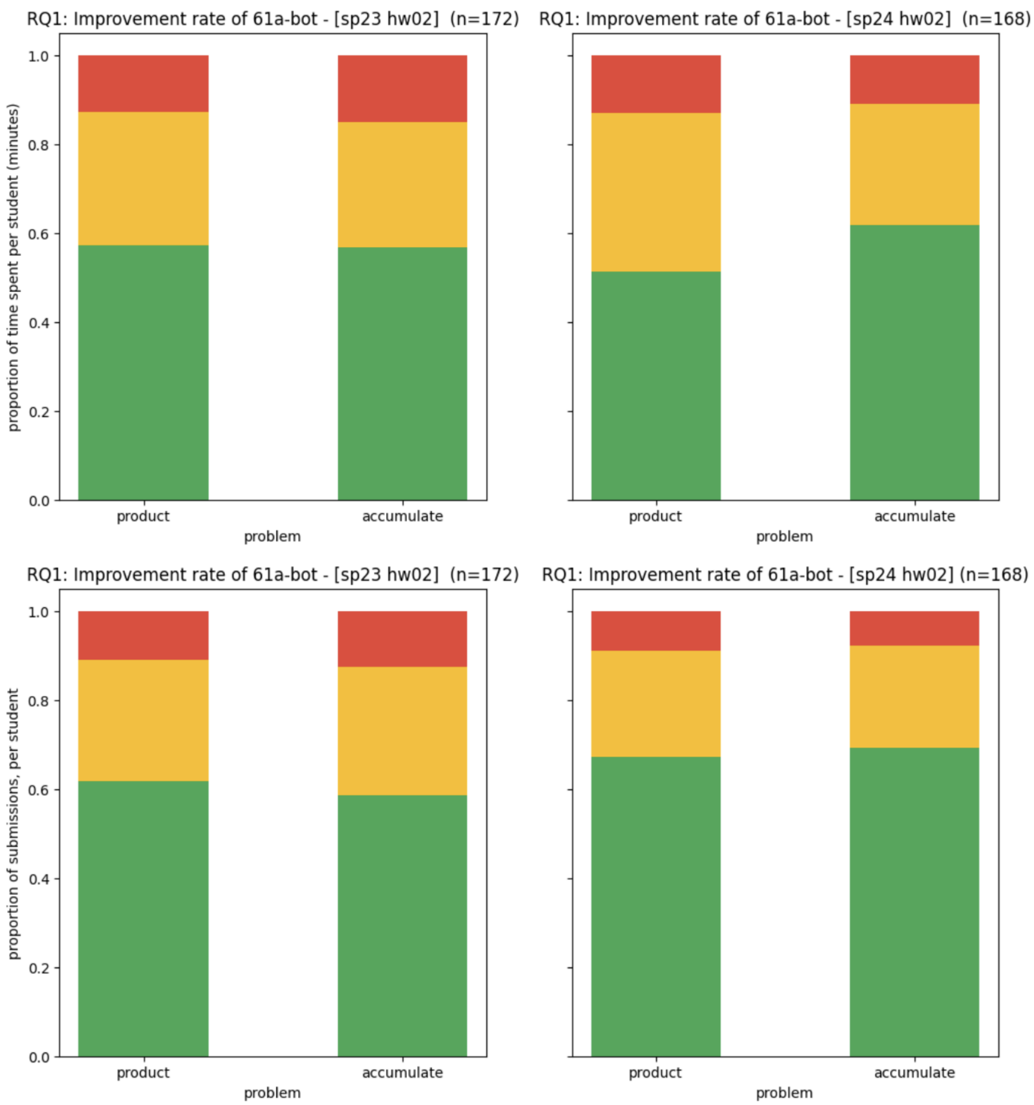

Figure C.24: Sp23 (Left) vs Sp24 (Right) HW2 Proportion of Time Spent & Number of Submissions Colored by Student Progression. Upper plots show average time spent per student. Lower plots show average number of submissions per student.

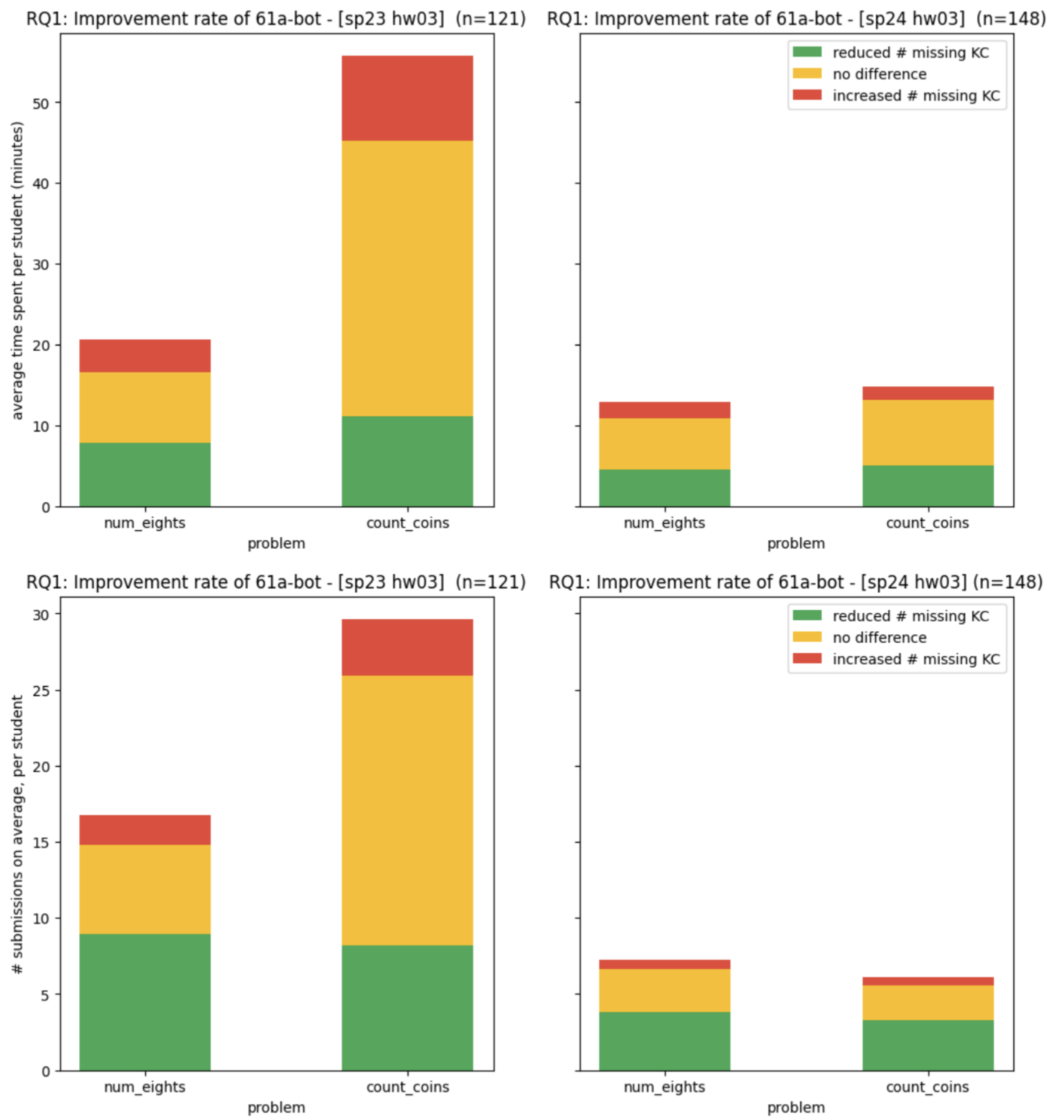

Figure C.25: Sp23 (Left) vs Sp24 (Right) HW3 Time Spent & Number of Submissions Colored by Student Progression. Upper plots show average time spent per student. Lower plots show average number of submissions per student.

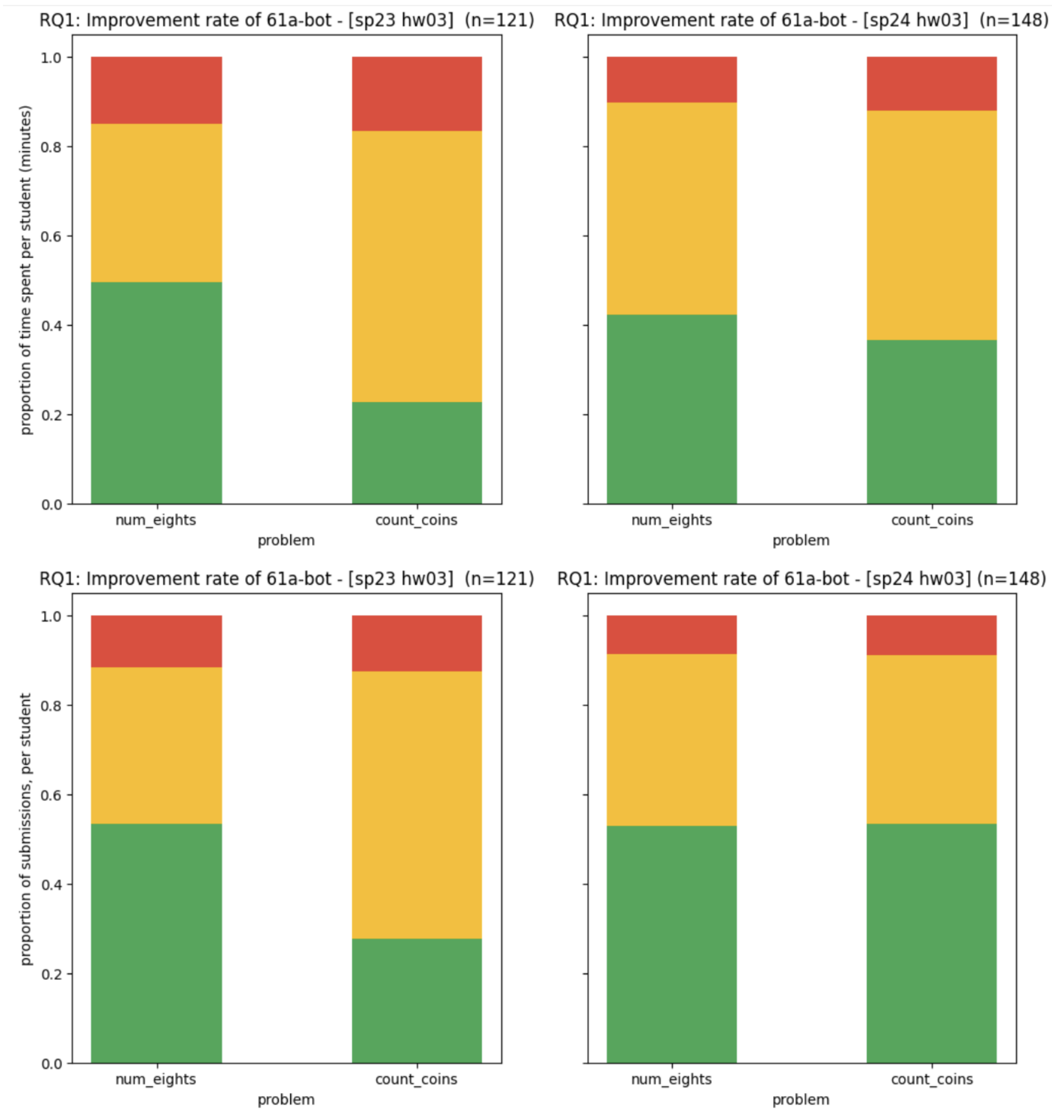

Figure C.26: Sp23 (Left) vs Sp24 (Right) HW3 Proportion of Time Spent & Number of Submissions Colored by Student Progression. Upper plots show average time spent per student. Lower plots show average number of submissions per student.

C.5 RQ2 Plots

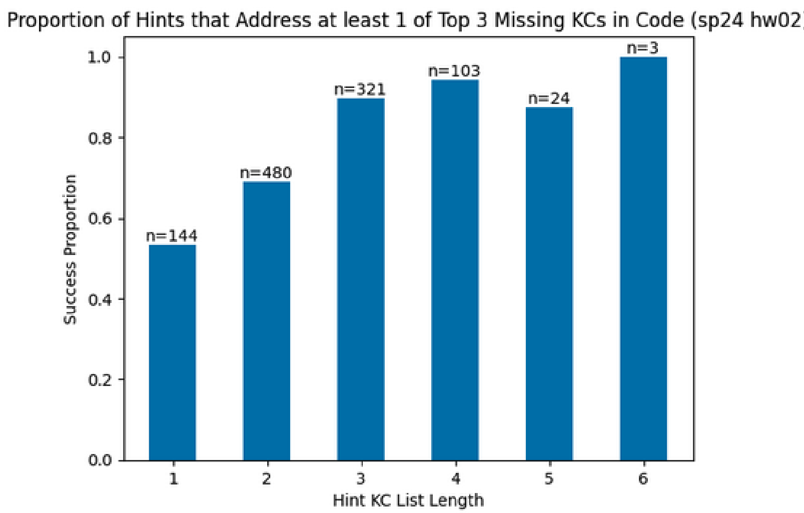

Figure C.27: Sp24 HW2 Proportion of Hints that Address at least 1 of Top 3 Missing KCs in Code Grouped By Hint KC List Length

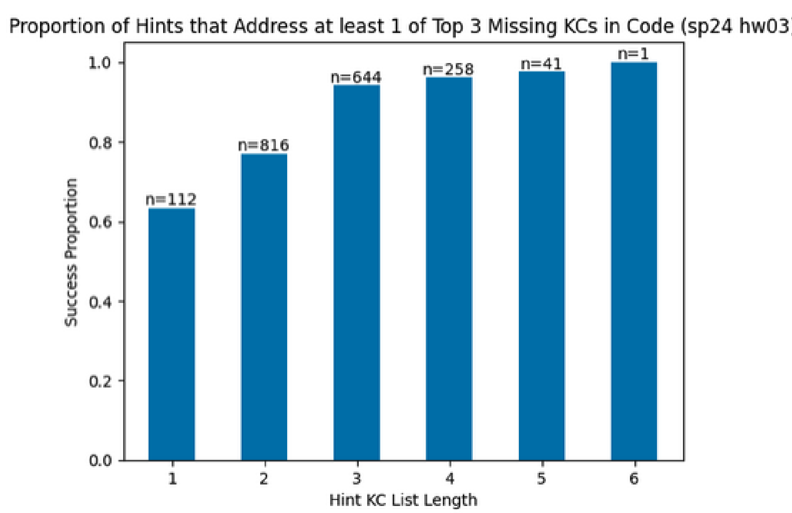

Figure C.28: Sp24 HW3 Proportion of Hints that Address at least 1 of Top 3 Missing KCs in Code Grouped By Hint KC List Length

C.6 RQ3 Plots

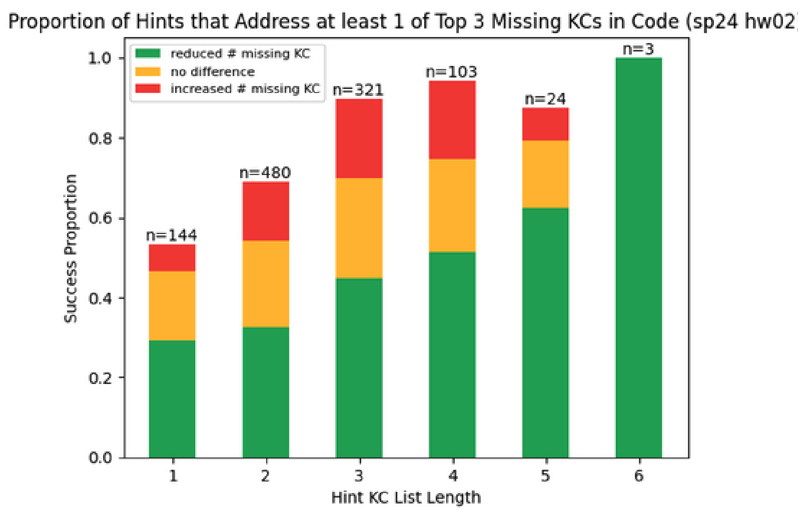

Figure C.29: Sp24 HW2 Proportion of Hints that Address at least 1 of Top 3 Missing KCs in Code Colored by Student Progression and Grouped By Hint KC List Length

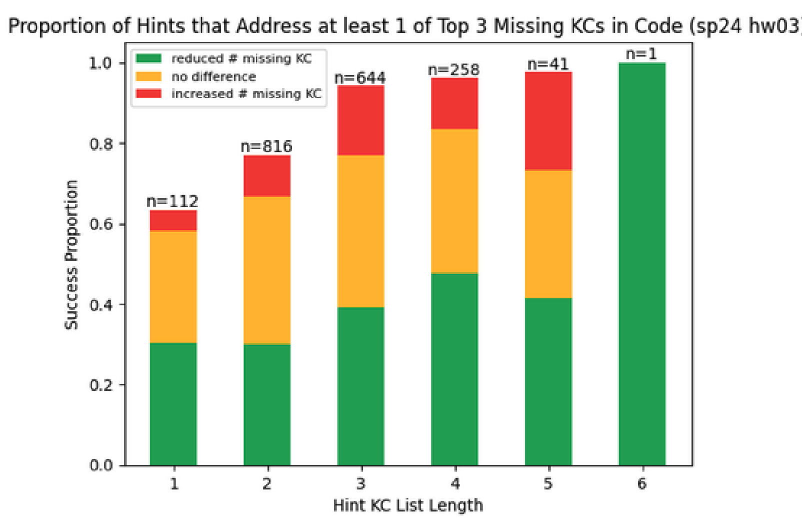

Figure C.30: Sp24 HW3 Proportion of Hints that Address at least 1 of Top 3 Missing KCs in Code Colored by Student Progression and Grouped By Hint KC List Length

Appendix D

Knowledge Component Lists

D.1 HW1 Knowledge Components

Homework problem statements can be found at `https://inst.eecs.berkeley.edu/~cs61a/sp23/hw/hw01/`.

```
Function Objects versus Function Calls: The student does not understand the
   difference between function objects and function calls. Function objects
    are data while function calls are expressions.

Call Expressions: The student fails to call a function properly. Call
   expression syntax in Python is `operator(operand1, operand2...)`.

While Loop Condition: The student has a logical error in their while loop
   condition.

Return Early in Loop: The student has a return statement in their while loop
    that causes it to end early.

Forget Return: The student forgets to return a value at the end of the
   function.

Return versus Print: The student misunderstands the difference between
   return and print. Return values become the value of a call expression
   while prints are used to display values in the terminal.

Boolean Operator Logic: The student doesn\textquotesingle t understand the
   evaluation rules of Python boolean operators including `==`, `!=`, `or`,
    `and`, `not`.
```

Forget While Loop Update: The student forgets to update their looping variable in their while loop, leading to an infinite loop.

Infinite Loop: The student\textquotesingle s code loops forever.

Problem Solving/Approach: The student\textquotesingle s fundamental approach to the problem is flawed and needs to rethink the structure of their program.

Statements versus Expressions: The student misunderstands the difference between statements and expressions. Statements are used to update the environment, enter control structure flow, etc. while expressions evaluate to values.

Understanding the Problem: The student misunderstands what the problem is asking them to do.

Indentation: The student has incorrect indentation, leading to a ` SyntaxError`.

If Conditional Expression: The student has an incorrect conditional expression in their if/elif predicate.

Missing Variable(s): The student is missing a variable needed to store key information.

Missing Syntax: The student is missing key syntax symbols, leading to a ` SyntaxError`.

Lists: The student uses lists, which are out-of-scope for the course.

Indexing Iterables: The student uses iterables, which are out-of-scope for the course.

Sorting Lists: The student uses sorting iterables, which are out-of-scope for the course.

Ternary Operator: The student uses ternary if expressions, which are out-of-scope for the course.

Recursion: The student uses recursion, which is out-of-scope for the course.

```
For Loop: The student uses a for-loop, which is out-of-scope for the course.

Blank: The student\textquotesingle s submission is blank or exactly the same
    as the skeleton code.

Extra Prints: The student has extraneous print statements in their code.

Missing Prints: The student is missing essential print statements in their
   code necessary for passing all the doctests.

Wrong Data Type: The student is using the wrong data type in their code (e.g
   . using floats when ints are expected).

Off-by-One: The student\textquotesingle s code doesn\textquotesingle t
   account for the right number of iterations. Specifically, they have
   miscounted the number of iterations their while loop runs by 1.

Assignment Statement: The student forgets to use or misuses assignment
   statements to create or update variables.

Arithmetic Logic: The student has flawed arithmetic logic leading to
   numerical errors.

Correct: The student\textquotesingle s submission is correct and passes all
   the provided doctests.
```

D.2 HW2 Knowledge Components

Homework problem statements can be found at `https://inst.eecs.berkeley.edu/~cs61a/sp23/hw/hw02/`.

List is same as Appendix D.1 except appended with the following KCs:

```
Returning Function Object versus Function Call: The student does not
   understand the difference between returning a function object (often the
    case in higher-order functions) and returning the result of a function
   call.

Functions as Data: The student does not understand what it means for a
   function to be the value of a variable. In other words, they do not
   understand first-class functions. In computer science, a programming
   language is said to have first-class functions if it treats functions as
    first-class citizens. This means the language supports passing
   functions as arguments to other functions, returning them as the values
   from other functions, and assigning them to variables or storing them in
    data structures.

Value Accumulation in Loop: The student fails to update a variable in their
   while loop via assignment statement properly leading to failure to
   accumulate a final total value.

Lambda Functions: The student misuses or misunderstands lambda functions.
```

D.3 HW3 Knowledge Components

Homework problem statements can be found at `https://inst.eecs.berkeley.edu/~cs61a/sp23/hw/hw03/`.

List is same as Appendix D.2 except appended with the following KCs:

```
Incorrect Base Case: The student either has an incorrect base case
   conditional or returns the wrong expression in the base case clause.

Missing Base Case: The student forgets an essential base case to consider.

Incorrect Recursive Call Location: The student attempts to make a
recursive call in the wrong place in their code.

Incorrect Recursive Call: The student passes the wrong argument(s) into
   their recursive call(s). In formal mathematical terms, they have the
   wrong inductive hypothesis.

Incorrect Recursive Recombination Step: The student does not the result of
   their recursive call correctly. In formal mathematical terms, they have
   the wrong inductive step.

Using Iteration instead of Recursion: The student tries to use iteration
   where recursion is expected and natural.

Incorrect Digit Manipulation: The student uses `%` (modulo) and/or `//` (
   floor-division) incorrectly in an attempt to parse the last and rest of
   the digits respectively.

Incorrect Helper Function Parameters: The student either calls their helper
   function incorrectly, calls their helper function with the wrong
   arguments, and/or chooses the wrong/wrong number of parameters for their
    helper function when defining it.

Missing Helper Function: The student is missing a helper function that is
   essential and natural to solving the problem recursively.

Incorrect Number of Recursive Calls: The student makes too few or too many
   recursive calls in their tree recursive function.
```

Appendix E

Student Progression Samples

The following samples are logs of student submission before-after pairs (along with the hint provided in between) categorized into one of the three categories described in § 5.1.

Homework problem statements can be found at https://inst.eecs.berkeley.edu/~cs61a/sp23/hw/hw01/.

E.1 Decreased Number of Missing KCs Sample (Green)

````
-----GREEN SAMPLE 1-----
<CODE-BEFORE @ 2024-01-21 10:34:02+00:00>:
```def hailstone(n):
    count = 1
    while n != 1:
        count += 1
        if n % 2 == 0:
            n = n // 2
        else:
            n = (n * 3) + 1
    print(count)
```
<CODE-BEFORE KCS>: Missing Prints, Return versus Print
<HINT>: Great job on implementing the `hailstone` function! However,
   remember that the function should also print out the hailstone sequence
   starting at `n`. You can achieve this by adding a print statement inside
    your while loop to print the current value of `n` before it is updated.
<HINT KCS>: Missing Prints
<CODE-AFTER @ 2024-01-21 10:45:20+00:00>:
```def hailstone(n):
    count = 1
````
````

````
    while n != 1:
        print(n)
        count += 1
        if n % 2 == 0:
            n = n // 2
        else:
            n = (n * 3) + 1
    print(n)
```
<CODE-AFTER KCS>: Forget Return

-----GREEN SAMPLE 2-----
<CODE-BEFORE @ 2024-01-25 11:43:55+00:00>:
```def hailstone(n):
    count = 0
    while n != 1:
        print(n)
        if n % 2 == 0:
            n = n // 2
        else:
            n = n * 3 + 1
        count += 1
        print (n)
    return(count)
```
<CODE-BEFORE KCS>: Off-by-One, Extra Prints, Missing Prints
<HINT>: Great job! Your hailstone function looks correct now. However, you'
   re printing the value of `n` twice in each iteration of the loop. You
   might want to remove the second print statement inside the loop.
<HINT KCS>: Extra Prints
<CODE-AFTER @ 2024-01-25 11:44:56+00:00>:
```def hailstone(n):
    count = 0
    while n != 1:
        print(n)
        if n % 2 == 0:
            n = n // 2
        else:
            n = n * 3 + 1
        count += 1
    return(count)
```
````
````

````
<CODE-AFTER KCS>: Off-by-One, Missing Prints

-----GREEN SAMPLE 3-----
<CODE-BEFORE @ 2024-01-26 07:45:34+00:00>:
```def hailstone(n):
    result = 0
    print (n)
    while n != 1:
        if n % 2 == 0:
                n = n // 2
                print (n)
                result += 1
        else:
            n = n * 3 + 1
            print (n)
            result += 1
    return result()
```
<CODE-BEFORE KCS>: Function Objects versus Function Calls, Off-by-One
<HINT>: Great job! You're almost there. However, you're trying to return `
   result()` but `result` is not a function, it's a variable. You should
   return `result` instead of `result()`.
<HINT KCS>: Function Objects versus Function Calls
<CODE-AFTER @ 2024-01-26 07:46:10+00:00>:
```def hailstone(n):
    result = 0
    print (n)
    while n != 1:
        if n % 2 == 0:
                n = n // 2
                print (n)
                result += 1
        else:
            n = n * 3 + 1
            print (n)
            result += 1
    return result
```
<CODE-AFTER KCS>: Off-by-One

-----GREEN SAMPLE 4-----
<CODE-BEFORE @ 2024-01-21 06:53:50+00:00>
````

````
```def largest_factor(n):
    i = n - 1
    while n % factor != 0:
        i = i - 1
        factor = n
    return factor
```
<CODE-BEFORE KCS>: While Loop Condition, Assignment Statement, Arithmetic Logic
<HINT>: You're on the right track with the `largest_factor` function, but there seems to be a small mistake. You're using the variable `factor` before it's defined. Also, you're assigning `n` to `factor` inside the loop, which doesn't seem to be correct. You might want to start `factor` at `n - 1` and decrement it in the loop until you find a number that evenly divides `n`.
<HINT KCS>: Missing Variable(s), Assignment Statement
<CODE-AFTER @ 2024-01-21 06:55:34+00:00>
```def largest_factor(n):
    i = n - 1
    factor = n
    while n % factor != 0:
        i = i - 1
        factor = n / i
    return factor
```
<CODE-AFTER KCS>: Arithmetic Logic, Wrong Data Type
````

E.2 Same Number of Missing KCs Sample (Yellow)

````
-----YELLOW SAMPLE 1-----
<CODE-BEFORE @ 2024-01-25 11:17:19+00:00>:
```def hailstone(n):
    count = 0
    while n > 1:
        if n % 2 == 0:
            n = (n / 2)
            count = count + 1
        else:
            n = (3 * n) + 1
            count = count + 1
    return count
```
<CODE-BEFORE KCS>: Off-by-One, Missing Prints, Wrong Data Type
<HINT>: Great job updating the value of `n` in your loop! However, you're
   missing one key part of the problem: you need to print out the hailstone
    sequence. This means you should print the value of `n` in each
   iteration of the loop, before you update it. Try adding a print
   statement at the beginning of your loop to print `n`.
<HINT KCS>: Missing Prints
<CODE-AFTER @ 2024-01-25 11:18:30+00:00>:
```def hailstone(n):
    count = 0
    while n > 1:
        if n % 2 == 0:
            print (n = (n / 2))
            count = count + 1
        else:
            print (n = (3 * n) + 1)
            count = count + 1
    return count
```
<CODE-AFTER KCS>: Missing Syntax, Missing Prints, Wrong Data Type

-----YELLOW SAMPLE 2-----
<CODE-BEFORE @ 2024-01-24 09:35:30+00:00>:
```def largest_factor(n):
    lf = 1
    while (lf<=n):
````
````

````
        if (n % lf ==0):
            n = lf
            lf=lf + 1
    return lf
```
<CODE-BEFORE KCS>: Problem Solving/Approach, While Loop Condition,
   Assignment Statement
<HINT>: Your current approach for the `largest_factor` function is not quite
    correct. You're on the right track with using a while loop and the
   modulus operator to check for factors, but you're changing the value of
   `n` inside the loop, which you shouldn't do. Instead, you should start `
   lf` from `n-1` and decrement it in each iteration until you find a
   number that evenly divides `n`. Can you try to modify your code with
   this approach?
<HINT KCS>: Problem Solving/Approach, Forget While Loop Update
<CODE-AFTER @ 2024-01-24 09:42:54+00:00>:
```def largest_factor(n):
    lf = 1
    while (lf<=n):
        if (n % lf == 0):
            x = lf
            lf=lf + 1

    return x
```
<CODE-AFTER KCS>: Infinite Loop, Forget While Loop Update, Arithmetic Logic

-----YELLOW SAMPLE 3-----
<CODE-BEFORE @ 2024-01-22 07:44:15+00:00>:
```def largest_factor(n):
    factor = n-1
    while n % factor != 0:
        factor = factor-1
```
<CODE-BEFORE KCS>: Forget Return
<HINT>: Great job! Your `largest_factor` function now correctly continues to
    check smaller numbers until it finds a factor of `n`. However, it seems
    like you forgot to return the factor once you find it. Remember to
   return the factor after the while loop.
<HINT KCS>: Forget Return
<CODE-AFTER @ 2024-01-22 07:46:47+00:00>:
```def largest_factor(n):
````

````
    factor = n-1
    while n % factor != 0:
        factor = factor-1
        return factor
```
<CODE-AFTER KCS>: Return Early in Loop
````
````

E.3 Increased Number of Missing KCs Sample (Red)

````
-----RED SAMPLE 1-----
<CODE-BEFORE @ 2024-01-25 12:55:58+00:00>:
```def hailstone(n):
    counter = 1
    while n != 1:
        counter += 1
        if n % 2 == 0:
            n = n//2
            print(n)
        elif n % 2 == 1:
            n = n *3 + 1
            print(n)
    return counter
```
<CODE-BEFORE KCS>: Missing Prints, Off-by-One
<HINT>: Great job! Your `largest_factor` function looks correct now. Also,
   your `hailstone` function seems to be on the right track. However,
   remember to print the initial value of `n` before entering the loop. Can
    you add that to your code?
<HINT KCS>: Missing Prints
<CODE-AFTER @ 2024-01-25 12:56:32+00:00>:
```def hailstone(n):
    counter = 1
    while n != 1:
        counter += 1
        print(n)
        if n % 2 == 0:
            n = n//2
            print(n)
        elif n % 2 == 1:
            n = n *3 + 1
            print(n)
    return counter
```
<CODE-AFTER KCS>: Extra Prints, Off-by-One, Missing Prints

-----RED SAMPLE 2-----
<CODE-BEFORE @ 2024-01-25 06:15:51+00:00>:
```from operator import add, sub
````
````

~~~~
def a_plus_abs_b(a, b):
    if b < 0:
        f = sub(a,b)
    else:
        f = add(a,b)
    return f(a, b)
```
<CODE-BEFORE KCS>: Function Objects versus Function Calls
<HINT>: You're very close! However, `f` should be assigned to the function `
   sub` or `add`, not the result of a function call. Try this: if `b < 0`,
   assign `f` to `sub`, otherwise assign `f` to `add`. Then, `f(a, b)` will
    be called in the return statement.
<HINT KCS>: Function Objects versus Function Calls
<CODE-AFTER @ 2024-01-25 06:20:44+00:00>:
```from operator import add, sub
def a_plus_abs_b(a, b):
    if b < 0:
        f = sub(f)
    else:
        f = add(f)
    return f(a,b)
```
<CODE-AFTER KCS>: Function Objects versus Function Calls, If Conditional
   Expression

-----RED SAMPLE 3-----
<CODE-BEFORE @ 2024-01-22 08:21:50+00:00>:
```from operator import add, sub
def two_of_three(i, j, k):
    return add( min(i,j,k) * min(i,j,k), min(j,k) * min(j,k))
```
<CODE-BEFORE KCS>: Problem Solving/Approach, Arithmetic Logic
<HINT>: Your `two_of_three` function is still not correct. The problem is
   that you're not always getting the two smallest numbers. For example, if
    `i` is 1, `j` is 2, and `k` is 3, your current function would return
   `1*1 + 2*2 = 5`, which is correct. But if `i` is 3, `j` is 1, and `k` is
    2, your function would return `1*1 + 1*1 = 2`, which is not correct.
   You need to find a way to exclude the largest number from the
   calculation.
<HINT KCS>: Understanding the Problem, Arithmetic Logic
<CODE-AFTER @ 2024-01-22 08:29:36+00:00>:
```from operator import add, sub
~~~~

```
def two_of_three(i, j, k):
    return add( min(i,j,k)**2, min(i,j,k) > min(i,j,k)**2)
```
```
```

```
<CODE-AFTER KCS>: Problem Solving/Approach, Arithmetic Logic, Missing
    Variable(s)
```