Exploring 'reverse-tracing' Questions as a Means of Assessing the Tracing Skill on Computer-based CS 1 Exams

Mohammed Hassan
mhassan3@illinois.edu
University of Illinois
Urbana, Illinois, USA

Craig Zilles
zilles@illinois.edu
University of Illinois
Urbana, Illinois, USA

ABSTRACT

In this paper, we perform a comparative analysis using a within-subjects 'think-aloud' protocol of introductory programming students solving tracing problems in both paper-based and computer-based formats. We demonstrate that, on computer-based exams with compiler/interpreter access, students can achieve significantly higher scores on tracing problems than they do on similar paper-based questions, through brute-force execution of the provided code. Furthermore, we characterize the students' usage of machine execution as they solve computer-based tracing problems.

We, then, suggest "reverse-tracing" questions, where a block of code is provided and students must identify an input that will produce a specified output, as a potential alternative means of assessing the same skill as tracing questions on such computer-based exams. Our initial investigation suggests correctly-designed reverse-tracing problems on computer-based exams more closely track a student's performance on similar questions in a paper-based format. In addition, we find that the thought process while solving tracing and reverse-tracing problems is similar, but not identical.

CCS CONCEPTS

• **Social and professional topics → Computing education**.

KEYWORDS

tracing, reverse-tracing, computer exams, CS 1

ACM Reference Format:
Mohammed Hassan and Craig Zilles. 2021. Exploring 'reverse-tracing' Questions as a Means of Assessing the Tracing Skill on Computer-based CS 1 Exams. In *Proceedings of the 17th ACM Conference on International Computing Education Research (ICER 2021), August 16–19, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3446871.3469765

1 INTRODUCTION

Computer-based exams have a long history in computer science. Often referred to as "lab exams", they were first proposed as a more authentic means of evaluating student code writing skills, because they permitted students to compile, test, and debug their code [2, 10, 25]. More recently, the rapid increase of students into computer science courses has led some universities to adopt computer-based exams to facilitate the auto-grading of code writing questions [37, 44], as well as address the logistics of running exams in large classes [16, 53, 55]. Online classes frequently rely on computer-based exams, including the recent shift to online teaching due to Covid-19. CS students often prefer computerized-assessments over paper-assessments mainly for the potential flexibility to choose an exam time [54], the ease of typing faster to edit code (compared to pen and paper) [41], as well as the auto-correction of syntax, unit tests, and debuggers from Integrated Development Environments (IDEs) [1, 53].

In most disciplines, the effect of shifting exams from paper to computer is minimal [8, 39], but previous work has noted significant format effects in computing courses when students can compile and execute code in the computer-format. Corley *et al.* and Lappalainen *et al.* found that CS1 student performance on code writing tasks did not significantly differ between the two formats in terms of understanding, but did differ significantly in syntax error frequency [11, 30]. In contrast, Grissom et al. found that CS2 students in the paper-format group of their study performed significantly worse in terms of understanding [20]. This prior research, however, has focused entirely on code writing questions.

In this paper, we focus on code tracing problems. As shown in Figure 1a, code tracing problems ask students to execute a provided fragment of code in their head, with the pedagogical goal of assessing a student's understanding of a programming language's semantics. While widely used on paper exams, code tracing problems are potentially problematic on computer-based exams, if students can execute the provided code to determine its output without needing to comprehend the code. In fact, in our study we found that there was effectively no correlation between student's ability to complete code tracing questions in the computer-based format and their ability complete them in the paper-based format. As shown on Figure 2, students always had a 100% performance on tracing problems in the computer-based format, compared to scoring from 33% to 100% on paper, a result discussed in Section 4.1.

One solution to the brute-forcing of code tracing problems is to prevent students from accessing software that permits code execution. Tools and products like the Safe Exam Browser, ProctorU, and Proctorio provide means for controlling the software to which a student has access during a computer-based exam. There are, however, a number of practical issues with using such software. First, faculty may desire to include both questions where such software shouldn't be accessed (e.g., code tracing) and questions where such software is allowed (e.g., code writing) on the same exam, increasing the burden on such an approach. Second, it may

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICER 2021, August 16–19, 2021, Virtual Event, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8326-4/21/08…$15.00
https://doi.org/10.1145/3446871.3469765

a)

```
def list_func(li, x):
    for i in range(len(li)):
        if li[i] > x:
            return i
    return -1

rval = list_func([0, 13, 22, 4, 31], 17)
```

What is the value of rval after the code above executes?

rval = [ ]

b) Input a list of at least 3 elements for `arg_list` such that `rval` is 3 after the code executes.

arg_list = [ ]

```
def list_func(li, x):
    for i in range(len(li)):
        if li[i] > x:
            return i
    return -1

rval = list_func(arg_list, 17)
```

Figure 1: Example tracing (a) and reverse-tracing (b) questions. Tracing questions are intended to assess whether students can execute code in their head.

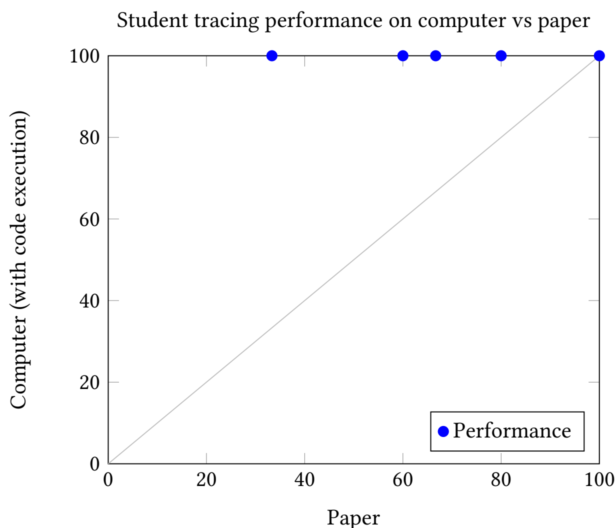


Figure 2: With access to code execution on a computer, subjects could always solve code tracing problems in the computer-format, independent of their ability to complete them on paper.

be not practical or worth the students' animosity to use such a heavy handed approach [35].

An alternative approach would be to identify a different question formulation that assesses the same skill as code tracing questions, but does so in a manner that is less susceptible to brute-forcing with code execution. To this end, we explore what we refer to in this paper as "reverse-tracing" problems. Reverse-tracing problems (as shown in Figure 1b) ask students to select an input value for a provided fragment of code so that it results in a specified output after the given code is executed. Such problems cannot be simply executed to find an answer, as doing so will result in an undefined variable if the student doesn't propose a candidate answer.

In this paper, we explore to what degree reverse-tracing problems can act as substitutes for tracing problems on computer-based exams. Specifically, we consider the following research questions:

- RQ1. How do students use code execution when solving tracing problems?
- RQ2. How do student problem solving methods for tracing problems compare to those for reverse-tracing problems on paper?
- RQ3. How do students use code execution when solving reverse-tracing problems?

To explore these questions, we performed a series of within-subjects 'think-aloud' interviews where students answered tracing and/or reverse-tracing questions in both computer- and paper-formats while vocalizing their thoughts. Our research methods are presented in Section 3. Our findings (Sections 4 and 5) can be briefly summarized as follows:

1. Students executed code on tracing problems in both what we view as productive ways (e.g., checking answers, exploring semantics) and in ways that circumvent the intent of the problem (e.g., mindless execution of the given code).
2. Solving reverse-tracing questions on paper has both similarities and differences to tracing questions.
    - For smaller and non-iterative code fragments, it appears that largely the same skill set is being used, but students construct a specification for the desired answer before selecting a specific answer.
    - For more complex code, especially with non-trivial loops, solution strategies diverge. Whereas book-keeping is the most common strategy for complex tracing problems, the students who successful solved complex reverse-tracing problems often constructed a high-level understanding of the code that allowed them to identify candidate answers.
3. While students used code execution in the same productive ways on reverse-tracing problems, the circumvention strategies were different. We observed both:
    - random guessing, particularly around small positive integers and features of the provided code.
    - pattern recognition of how the output related to the input that didn't involve understanding the code

Overall, we found that students' ability to complete reverse-tracing questions on the computer was more correlated to their ability to do so on paper than was true of code tracing problems.

That said, in our collection of problems, there were problems that were susceptible to circumvention strategies (i.e., some students got the correct answer without understanding the code). In the discussion (Section 6), we suggest principles to question design that can partially mitigate these strategies.

2 RELATED WORK

2.1 Exam mode: Paper vs. Computer

A few studies found that student performance between paper and computer assessments significantly differed in some code writing problems, potentially introducing unwanted advantages or disadvantages to students. Corley *et al.* found that CS 1 students' problem solving ability was largely unaffected by the different assessment formats in code writing tasks, and that student ability to correctly write syntax was significantly worse in the paper-and-pencil version [11]. Similarly, Lappalainen *et al*'s study found that CS1 students in both paper and computer groups had similar performances in terms of understanding on a Rainfall code writing problem, but the paper group had more syntax and minor cosmetic errors (e.g., misspelled variable name) [30]. In contrast, Grissom *et al.* found that CS 2 students' solutions demonstrated higher levels of understanding in the computerized formats when writing implementations of recursive binary search trees [20]. As students in the computer-format groups had access to Integrated Development Environments (IDEs) [11, 30], it is not surprising that students had fewer syntax errors, because IDEs can highlight and auto-correct incorrect syntax, but the mixed results related to understanding/problem solving are harder to explain. Computer-based exams can also benefit instructors with auto-grading and automatic analysis of students' learning progress [45].

Prior work has explored students struggles with learning to use IDEs, using both qualitative and quantitative methods [5, 6]. Students' struggles with IDEs provides some insight to how computerized formats for code writing problems could have a negative effect compared to paper-formats.

It is also noteworthy that past studies that evaluated the performance difference between computerized and paper assessments only evaluated the product of student work and not the process in which it was generated [11, 20, 30], in contrast to the current work. Also, these previous studies evaluated the differences only on code writing problems, while our study focuses on code tracing (and reverse-tracing) problems. We expect that a shift to computer-based exams has a qualitatively different impact on code tracing problems (compared to code writing, Parson's problems, and "Explain in plain English" problems), because the other common CS 1 problem types are not as susceptible to brute-force machine execution.

2.2 Code Tracing

Past studies have found that if students can trace, or hand-execute, code then they are more likely able to read code (understand code by reading it without manual execution) and are also more likely able to write code. Tracing tends to be a precursor to reading [46] and both tracing and reading tend to be precursors to writing [23, 28, 29, 32, 33, 46, 48], making it important to ensure that our students are developing the foundational skill of code tracing.

Past qualitative think-aloud studies of tracing on paper found that students who correctly kept track of the values of variables as they change per line of code in tracing problems tend to perform better than students who do not. These studies found students writing the values of variables and their changes each iteration on paper [13, 14, 31, 52]. Cunningham *et al.* found that the reason students kept track of variables is due to having too many variable dependencies and arithmetic in the code. Students also tended to keep track of variables if they traced code incorrectly initially [14]. Vainio *et al.* found students using an incorrect tracing strategy, "single value tracing," where students kept track of the most recently assigned value of any variable to at most one variable [47]. To the best of our knowledge, there are no studies related to completion of code tracing questions on computer-based assessments.

While the most common tracing errors are related to incorrect (or the lack of) variable bookkeeping, other errors in tracing exist, such as (1) arithmetic errors [14], (2) guessing based on an incorrect but common pattern that doesn't apply to the specific problem [19], (3) incorrect order of execution [26], (4) spending too much time on irrelevant parts of code and then running out of time, and (5) making incorrect assumptions about programming language concepts (e.g., `for` loops repeating the entire program instead of only what is inside its block) [47].

2.3 Beyond Line-By-Line Tracing: Higher-level Reading

Brooks proposed a top-down model towards program comprehension, where the programmer creates an initial hypothesis of the program based on their domain knowledge (real-world knowledge of the problem the program aims to solve) and their programming language knowledge [7]. The specificity of this initial hypothesis varies across programmers and domains, but these initial hypotheses typically cover at least the inputs, outputs, major data structures, and primary processes not including implementation-level details. Then, the programmer creates subsidiary hypotheses to explore specific details needed to validate the parent hypothesis, and this process hierarchically repeats until subsidiary hypotheses cover the implementation-level details. The programmer may also compare and contrast any subsidiary hypotheses to alternatives to refine their understanding of the program.

Complementing Brooks' model, Pennington proposed a bottom-up model where if the programmer is unfamiliar with the problem the code solves (e.g., fails to create an initial hypothesis), they create a lower-level control flow model first [38]. In this model, they group together chunks of code based on their control flow relationships. From there, they create a higher-level data-flow model (inputs, output, functions) where they can then integrate domain knowledge. Sub-goals are chunked together towards creation of the data-flow models. Throughout our interviews of our more complex reverse-tracing problems, we see programmers employ a mix of both Brooks' top-down and Pennington's bottom-up models, where the programmer starts with a low-level control flow analysis (constraints), and then states a higher-level hypothesis description of an answer (e.g., Section 5.2.3).

Detienne *et al.* define two strategies for comprehending programs: symbolic simulation and concrete simulation [17]. Symbolic

simulation was used initially by all participants in their study, where participants attempt to connect code to a prior learned "plan" (or stereotypical action sequences). If they are unable to apply a plan to the code (e.g., the code is unfamiliar, or the closest matching plan does not align well enough), then participants may resort to the concrete simulation strategy. Concrete simulation is a line-by-line mental execution of the program, with the goal to verify that prior learned plans align in all interactions (e.g., all possible inputs). They also define other strategies, such as reasoning according to "rules of discourse" (i.e., using one's practical knowledge about how code is generally written) and "principles of the task domain" (i.e., non-programming knowledge about the task's context) in the situation that no plan aligns with the code to create a new plan. We have one instance where a participant applied symbolic simulation on one of our problems (e.g., Section 4.3), and other instances where many participants applied concrete simulation (e.g., Section 5.2.1).

Gugerty *et al.* studied how novice and expert programmers modified code to fix a bug. Both novices and experts begin by reading the code and making an initial modification to the code in an attempt to fix the bug. Novices often had an initial incorrect fix (modification) and created more bugs in the code, contrary to experts who often had an initial correct fix. This may demonstrate that experts often created a correct initial hypothesis of the code after reading it, because they had an idea of what the bug in the code was and proposed a fix [21]. In an eye-tracking study, Busjahn *et al.* found that novices read code more linearly (top-to-bottom, left-to-right, like reading English text) than experts, suggesting that experts are more strategic in how they read code to comprehend programs (e.g., following execution order and relevant parts of code) [9]. We see examples of both linear and non-linear (e.g., execution order) reads of code in our problems (e.g., Sections 5.2.1 and 4.3 respectively).

In another eye-tracking study, Crosby *et al.* found that experts (moreso than novices) identified and utilized beacons to understand programs. Beacons include code comments, descriptive identifiers revealing the program's purpose (e.g., variable and class names), and the lines of code that are central to the algorithm of the program (e.g., the most informative line/lines of code) [12]. Bednarikn *et al.* conducted a debugging, eye-tracking study where novice and expert programmers were given a diagram (UML) representation of the code in addition to the code. They found that novices often relied on the diagram representation, and were often unable to fix bugs in code after later reading the output of the code. Experts, on the other hand, relied mostly on just the code and the output, and systematically connected parts of the code to the output [3]. In some of our interviews, participants used the online IDE, `repl.it`, to view the outputs of our reverse-tracing code, and then reasoned based on the output (e.g., Section 5.3.2).

2.4 Alternative Question Formats to Tracing & Higher-Level Thinking

Izu *et al.* asked students to evaluate whether four program transformations were equivalent (e.g., all possible inputs leads to same corresponding output). Relative to ordinary tracing questions, this type of question seemed to encourage students to think at a higher-level, classified based on the SOLO Taxonomy relational level [4, 15] (purpose of a program in plain English), and the block model above the atomic level [42] (ranging from the main purpose of a program to analyzing relationships between lines of code and different execution paths) [24]. Similar to their findings, our more complex reverse-tracing questions also seemed to elicit higher-level explanations from students (e.g., Section 5.2.3), while our simpler reverse-tracing questions had many instances of line-by-line hand-execution (Section 5.2.1).

In another study, Izu *et al.* asked students to evaluate whether different programs can be "reversed," which means that the code can be modified such that it restores the value of variables to a previous state (e.g., output of program changes back to value of input). The concept of finding whether code is reversible involves finding an "overlap" of inputs, which means at least two different inputs lead to the same output [34]. Many of our findings of our reverse-tracing problems also have overlaps in inputs in the form of input specifications (multiple possible inputs to get a specified output), which are highlighted in Sections 5.2.2 and 5.2.3.

While we do not claim to have invented reverse-tracing problems, we are not aware of any previous research literature related to them.

3 METHODS

We conducted a series of think-aloud studies to observe the process students (the participants) use to solve code tracing and reverse-tracing problems across paper- and computer-formats.

3.1 Participants

This paper reports on the analysis of 13 think-aloud sessions.[1] The participants were undergraduate students recruited from a Python-language introductory programming course for non-computer science majors at a large public U.S. university. The participants included nine males and four females and were traditional aged undergraduate students (e.g., age 18-23). With IRB permission, we sent email to students completing the course in Spring and Fall 2020 to solicit volunteers. Participants consented to audio and screen recordings of the sessions. Participants were given $15 gift cards as compensation for participation in this nominally 1-hour study.

3.2 Process

The think-aloud sessions were done for each participant one-by-one. The sessions took place over Zoom video conferencing software due to COVID-19. Every participant was asked to complete a series of Python language tracing and/or reverse-tracing problems. In all interviews, approximately half of the problems were in the paper-based format and the other half were in a computer-based format, with the exception of one interview where the participant completed all computer-based questions but ran out of time after the first question on the paper-format. Five of the sessions had both tracing and reverse-tracing questions and are the focus of the results that are presented in Section 4. The remainder focused exclusively on reverse-tracing problems; all interviews contributed to the results in Section 5.

In the paper-based format, students typically positioned a cell phone to record what they wrote on paper or wrote on a tablet

[1] Additional sessions were conducted during which the study protocol was revised. Nothing from those sessions contradicts the findings presented here, but those sessions were not analyzed to the same depth.

screen shared to Zoom. Questions were shared by the interviewer to the participants via Google Docs; participants were asked to not use their computer for any purpose besides viewing the problem.

In the computer-based format, questions were provided either by Google Docs (early interviews) or through the PrairieLearn assessment platform [49, 50] used in the class the subjects were recruited from (later interviews). In addition, it was communicated to students verbally (early interviews) or through instructions on PrairieLearn (later interviews) that students were allowed to use the `repl.it` [40] web-based Python interpreter. In some interviews, the interviewer reminded participants that using `repl.it` was allowed. The interviewer tried to walk the fine line between ensuring that the participant knew that use of a Python interpreter was allowed without compelling its use by participants.

The questions covered a range of topics and difficulties. For each kind of question—tracing and reverse-tracing—we attempted to create two matched pools (A and B) of questions by constructing pairs of questions that had similar topic coverage and difficulty and assigning one to each pool. In large part, the reverse-tracing questions were generated from the tracing questions by choosing a desired output and selecting one input variable to leave undefined.

In order to mitigate any potential sequencing effects in the data, half of the subjects completed the paper-format first and the other half completed the computer-format first. For those subjects that had both tracing and reverse-tracing problems we also randomized the order of the two types of questions. Finally, across the subjects, we varied which pool (A or B) was used for the computer-format, with the other pool being used for the paper-format.

Throughout the whole session, participants were encouraged to verbalize their thought process. Our think-alouds followed the protocol of Ericsson *et al.* for recording unstructured verbalizations [18]. Their approach aims to minimize the extra cognitive effort required to verbalize thought processes out loud to help prevent any unknown and unwanted third factors from impacting student performance. Participants were only asked to say what was currently on their mind as they were solving the problems and were not asked to explain nor interpret their thought process for our benefit [18]. If participants were silent for more than 2-3 minutes, they were reminded to think-aloud. They were trained until they were comfortable with the process of thinking aloud.

Participants were given as much time as they needed to solve each problem, but if they appeared to be stuck on the problem for more than 2 minutes (appearing to make no progress), they were encouraged to try a different problem. After half of the interview was complete, they had the option of a 5-10 minute break. After the study was complete, the participants had the opportunity to ask any questions about the study.

3.3 Analysis

For the first five think-alouds, the audio portion of each recording was transcribed and composited with still images from the Zoom recording showing the computer screen or paper state at important events. The transcripts were inductively coded independently by two researchers in groups of 1-3 transcripts at a time. After coding each group of transcripts, the two researchers met to discuss the coding and reconcile differences in coding. They discussed and agreed on examples and counter-examples for new emerging categories.

The remainder of the think-alouds were coded directly from the video. Again, two researchers inductively coded the source material independently in groups of 1-3 videos. After coding each group of videos, the two researchers met to discuss and refine emergent themes.

4 RESULTS: TRACING

As previous work has focused on how students solve tracing problems on paper, we focus here on comparisons between paper and computer tracing, how students use code execution when solving tracing problems, and a technique we saw students use to solve tracing problems consistent to the behavior of novices at the Neo-Piagetian concrete operational stage in Teague *et al*'s study [46], as well as the "symbolic simulation" strategy in Detienne *et al*'s study [17].

4.1 Tracing performance: computer vs. paper

We analyzed five sessions where students worked on tracing problems. On paper, the performance of the four participants that attempted all problems varied from 33% to 80%. A fifth student, who ran out of time before completing all problems, correctly solved every problem that they attempted. In contrast, all of the students were able to compute correct answers for every problem when they had access to `repl.it`. The relationship between the student scores in each mode are plotted in Figure 2. While the number of students observed for tracing was small, it was sufficient to demonstrate that there is likely a weak correlation between students' ability to complete tracing problems on paper and on computer. As a result, we chose to focus the rest of the interviews solely on reverse-tracing problems.

While students can earn full credit on code tracing problems through computer-assisted code execution, they may not be able to do so as quickly as someone solving them through mental execution. Standard ways of presenting questions to students electronically (e.g., both PDF and in the browser) can be set up to prevent trivial copy-and-paste, so that students have to take the time to re-type the code to get the computer to execute it. This time penalty might be meaningful on a timed exam, but we do not attempt to characterize it here.

4.2 How students use `repl.it` on tracing problems

We observed three main behaviors of how participants chose to use `repl.it` while solving tracing problems. We characterize these as: (1) students checking answers that they discovered manually, (2) students using `repl.it` to verify specific semantic properties of Python, and (3) executing the code on `repl.it` before understanding it. We discuss each of these in the following subsections, but we make no attempt to characterize the relative frequency of each behavior. We strongly believe that the presence of the interviewer induced a social-desirability bias [27] in at least some of the participants that led them to behave differently than they would in an exam situation without someone watching their specific actions.

```
ain.py
1   x = "apple"
2   y = x.upper()
3
4   print(y)
```

```
APPLE
>
```

Figure 3: Student writes code in `repl.it` merely to test the behavior of the `string.upper` method.

4.2.1 Double-check a complete solution. This category covers instances when participants first attempt to completely solve the tracing problem without the aid of `repl.it`. They either provide a specific value for the output or a complete description of the solution where the researchers could easily infer a value. The description or value may be an incorrect solution, but displays a complete tracing attempt.

After determining an output, participants in these instances enter the code into `repl.it` in order to check their answers against the execution's output. While we would consider this, in principle, to be productive way to use code execution on tracing questions, if the student's answer was incorrect, the code execution provides the correct answer (bypassing the student having to figure out why their answer was wrong and what the correct answer should be).

4.2.2 Execute Code to Remember How Certain Constructs or APIs work. Examples in this category occur when participants appear to forget how certain language constructs (e.g., `return`) or built-in API methods (e.g., string `upper` method) behave in Python. In these instances, they execute code in `repl.it` that is either a small subset of or distinct from the question's given code, in order to remember how those constructs or methods work or how to use them.

For example, Participant 1 read the code from one question aloud, then appeared unsure of what the `string.upper` method did. They decide to check `upper` by testing a sample string rather than running the entire provided code fragment, as shown in Figure 3.

> If I really want to learn I could just test what `upper` does by putting the word `upper`. I think that's how `upper` works. And then find out what happens to that word. *(runs code)* Blaringly obvious. OK, so `upper` does capitalize every single letter. So now I know for a fact that I can. I can literally predict what's going to happen here without even running it.

This particular student may have chosen to execute a fraction of the code instead of the whole due to the aforementioned social-desirability bias. They express that they feel just executing the whole code fragment in `repl.it` is cheating.

> I feel like `repl.it` is just giving me the answers to everything even before I (pause) Like, I even say (pause) should I do that? 'cause I feel like I'm just cheating by putting it on `repl.it`. Like, it's just giving me the answers to everything.

Nevertheless, we consider this behavior of exploring the language syntax to be productive, as the student might gain knowledge during the exam.

4.2.3 Execution before seriously attempting to understand the code. Examples in this category involve instances where participants make no attempt or only half-hearted attempts to trace the code manually before copying the code to `repl.it`. Multiple participants verifying with the interviewer that this was allowed.

```
def f(li, h):
    x = 0
    for c in li:
        if h in c:
            x += 1
    return x

print(f(["John", "Mary", "Jackie", "Crab"], "J"))
```

Figure 4: Example tracing question that counts the number of strings with a given letter.

> **Participant 2**: So I can use [`repl.it`] in any way I want? Alright, I would honestly probably just copy and paste it and see what comes up. (executes code)

> **Participant 4**: What is the output when the following code snippet is run? Alright, so I can just copy this and find the output? (executes code)

After running the code, participants tried to explain to the interviewer why the answer received from `repl.it` was the correct one. In some cases, their explanations demonstrated an understanding of why the output was produced. In other cases, they were unable to provide a correct explanation in spite of having the correct output. In a few cases, students did not even attempt to explain the code. Unsurprisingly, we find this to be an undesirable use of code execution.

While one might expect that it would be the students that performed the weakest on the paper-based tracing questions that would be the most prone to skip trying to solve tracing questions manually and turn directly to `repl.it`, we saw no such pattern. On the contrary, **Participant 2 and 4** exhibited this behavior in nearly all of the computer-based tracing problems, but had the two highest scores of students that completed all of the paper-based questions. Use of this strategy by a participant during our interviews, however, may not be predictive of its usage in an actual exam situation.

4.3 Solving tracing problems through building a mental model

In a few cases, we saw students exhibit a behavior similar to novices operating at the Neo-Piagetian concrete operational stage described in Teague *et al*'s study [46]. Students at the concrete operational stage would abstractly trace code using sets/ranges of values rather than individual concrete values and/or reason about the constraints on/relationships between values (e.g., `x` must be greater than `y` on a path where `x > y` evaluated to true).

In one such instance, Participant 3 was reading the code shown in Figure 4. They read the code from the function invocation to the function definition, consistent to how experts read code non-linearly (e.g., execution order) in Busjahn *et al's* eye-tracking study [9]. As they read the code, they were recognizing constraints based on the code structure (datatype `string`, `if "J" in string` name), piecing together the overall purpose of the function.

> We are taking a list of strings—looks like they're names possibly—and then a single character. `x` (pause) Here we're doing it like a counting algorithm. So `for c in list`– So, for every single name– `if h` Okay, so if the letter "J" is inside the name, we're going to increment our count by one.

Once they have inferred the purpose of the function—what might be referred to as a relational understanding of the function in the SOLO taxonomy [4, 51]—they then apply their understanding of the function's purpose on the input data rather than stepping through the code line-by-line.

> Okay, so (referring to "`John`") 1, and (points to "`Mary`") no "`J`" in there. (points to "`Jackie`") 2. (non-verbally dismisses "`Crab`") So it's going to output 2, I think. Yeah.

This is also consistent with the findings of Detienne *et al* [17], in which the student performed "symbolic simulation" by recognizing and forming a "plan" of the code for solving the tracing problem, which is the "counting algorithm" plan. In hindsight, this might be due to the design of some of our tracing problems. We intended to design the code of our tracing problems without a purpose, so as to specifically assess the detailed book-keeping skill, but this example may be an unintended exception. We in part point out this behavior in tracing questions because we see a similar behavior in some of the more complex reverse-tracing questions, as discussed in Section 5.2.

5 RESULTS: REVERSE-TRACING

In this section, we shift our focus to reverse-tracing questions. We present data that suggests they are less susceptible to brute-force execution on computer than tracing questions, we characterize the tools that students use to solve them on paper, and discuss how students used `repl.it` when solving them on computers.

5.1 Reverse-tracing performance: computer vs. paper

For comparing the student performance on reverse-tracing questions in the computer-format and paper-formats, we use the eight most recently analyzed sessions. The earlier sessions only included the easier versions of reverse-tracing questions (as this question-format was newly introduced to the course at the time), so those participants all received perfect or near perfect scores from both modes. We later created reverse-tracing problems that matched the difficulty of our tracing questions. In contrast to the tracing questions, our participants on the computer-based reverse-tracing questions had a wide range of scores, as shown in Figure 5.

While we have far too few data points to compute a statistically meaningful correlation coefficient, qualitatively it can be seen that there is a much stronger correlation between formats for reverse-tracing problems than for tracing problems. For five of the students, the correlation is particularly strong. For the other three students, their performance on the computer-format questions is roughly 40% higher than the paper-format. In two cases, these students successfully employed the gaming strategies described in Section 5.3. In the third case, the student merely made careless errors in the paper-based questions; had the student made those errors on the computer-based questions they would have detected the mistake in `repl.it` and fixed their answer before submitting.

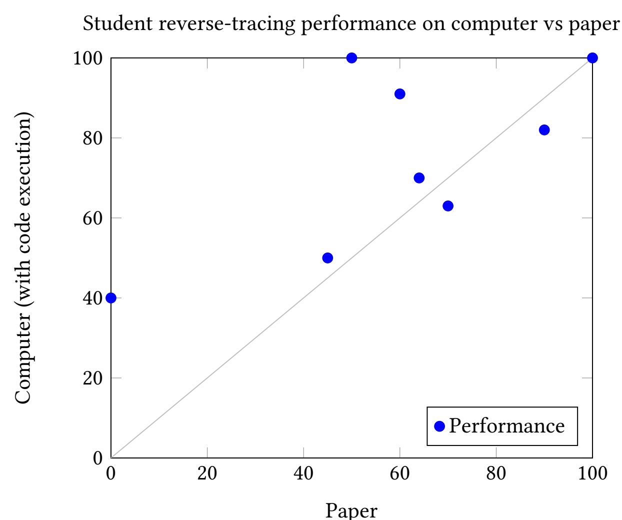


Figure 5: Access to code execution had less impact on students' ability to complete reverse-tracing problems.

What value should variable `y` be so that the variable `x` becomes the integer `16`?

```
x = 3
if x < 11:
    x += 5
if x == y:
    x *= 2
```

Figure 6: Simple, non-iterative reverse-tracing question focused on understanding of conditionals.

5.2 How Students Solve Reverse-Tracing Problems

In order to complement the extensive prior work on how students solve tracing problems [13, 28, 31–33, 46, 47, 52], we will attempt to characterize how participants solve reverse-tracing problems. In particular, this section will focus on how they were solved on paper, and, in Section 5.3, we'll present how subjects used `repl.it` as part of their solution process. We identified three main concepts about their solutions:

5.2.1 Smaller and non-iterative code fragments are analyzed similarly to tracing problems. When a reverse-tracing problem involves a relatively small amount of code, especially if it doesn't involve a loop, we find that the reasoning process has significant similarities to what is observed in tracing. Subjects consider the code one line at a time, computing the state of variables as they progress.

For example, the code shown in Figure 6 is only five lines long and was designed to test a student's ability to trace `if` statements, potentially before the course introduces loops. Participant 5 solves this problem by tracing the code forward, performing book-keeping

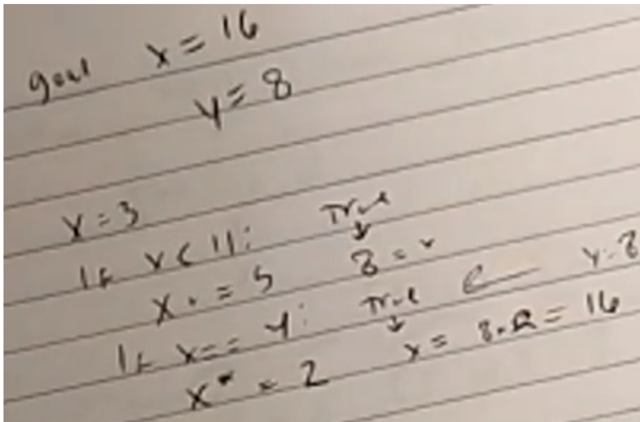
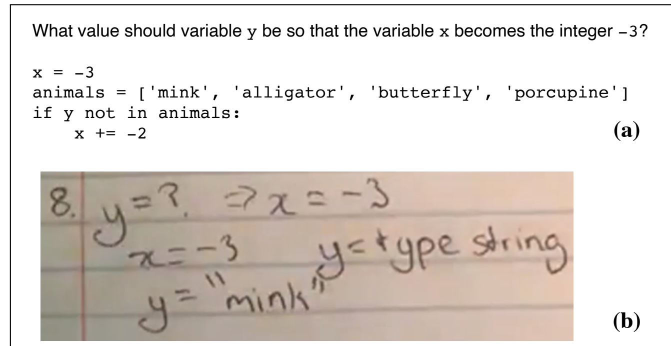


Figure 7: Participant 5 appears to sketch a line-by-line trace of the code, and wrote the correct value for the variable `y` that makes the second `if` block `True` to obtain the correct result of `x = 16`.

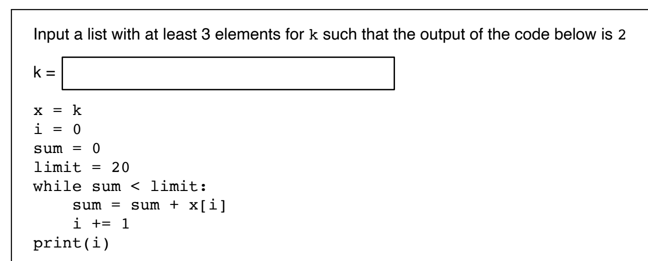

What value should variable y be so that the variable x becomes the integer -3?

```
x = -3
animals = ['mink', 'alligator', 'butterfly', 'porcupine']
if y not in animals:
    x += -2
```

(a)

(b)

Figure 8: A reverse-tracing question where there are many possible correct answers (a), and participant 5's notes when solving this question (b).


Input a list with at least 3 elements for k such that the output of the code below is 2

k = [ ]

```
x = k
i = 0
sum = 0
limit = 20
while sum < limit:
    sum = sum + x[i]
    i += 1
print(i)
```

Figure 9: A reverse-tracing question that is difficult to reverse-trace using bookkeeping. Successful students traced/read the code in a forward direction to build a mental model of it that they then use to identify a suitable input.

of the variable `x`'s value as they proceed. When they reach the second branch, they have to look ahead to see which branch outcome will lead to the desired result, but the code is short enough that this can be quickly identified and the desired direction of the branch is identified. A screen shot of the participant's written work is shown in Figure 7.

> The goal is to get `x` equal to 16. So `x` is 3. Because 3 is less than 11, we add 5 to `x`. So after the first if statement, `x` equal to 8. And then we say, if `x` is equal to `y` (pause) then we multiply `x` by two and `x` is set equal to that value 16. So I'm thinking that `y` would have to be equal to 8.

While the above example is largely traced in the forward direction, some pieces of code are more naturally traced in the reverse direction. While addition and subtraction are (largely) reversible operations, many operations are not, which motivates the second observation.

5.2.2 Reverse-tracing problems don't necessarily have a single answer; they require identifying answer specifications. For many operations, there are many possible inputs that result in a given output. For example, with Python's *floor division*, which rounds down to the closest integer, there are three values of `z` where `z // 3` will equal 5, namely 15, 16, and 17. Similarly, there are many strings that have a length of 13 characters and lists of integers that sum to 53.

So when tracing a statement backwards, it is not always possible to identify *the* input to a statement. Instead, we hear students describe what we'll refer to as *specifications* for values at particular points in the execution. They'll describe properties that values need to have. Once they reach the beginning of the computation, they'll select a specific value that meets the specifications and that will be their answer. This process is similar to the behavior that novices at the Neo-Piagetian concrete operational stage exhibit when solving tracing problems described in Teague *et al*'s study [46] (described above in Section 4.3).

For example, when participant 4 solved the problem in Figure 8a, they reasoned first that since `y` is being checked for membership in a list of strings that it must be a string. Then they reasoned that they wanted the `if` condition to be false, so that `x` would keep its value, and that specified that `y` needed to be one of the values in the list `animals`. The specification for `y` is successively refined until a particular value can be selected (e.g., "`mink`").

> `y` is a string. We want `y` to be in `animals` so that the next thing doesn't go in [and] `x` remains negative 3. So we would want `y` to be equal to possibly a string "`mink`".

5.2.3 Summarizing code to reverse-trace complex code. For complex code fragments, especially those containing loops, it can become difficult for our subjects to predict the flow of control backwards and keep straight the specifications for the variables. For these more complicated code fragments we see successful students shift strategies: they trace/read the code in a forward direction and attempt to build a mental model of the code which they can use to identify a suitable input.

Participant 11 demonstrates this process on the problem shown in Figure 9. From their verbalizations, one can see that they're piecing together what the code is doing bit by bit and then reasoning about what must be true about the input in order to achieve the

Input a value for `arg_num` such that `rval` is `2`.

arg_num = [ ]

```
def list_func(li, x):
    for i in reversed(range(len(li))):
        if li[i] == x:
            return i
    return -1

rval = list_func([1, 2, 6, 9, 10, 14, 16, 17], arg_num)
```

Figure 10: A reverse-tracing question that many participants struggled with because of `reversed(range(len(li)))`.

Input a list for `k` such that `k`'s value after the code below executes is `[5, 6, 5]`

k = [ ]

```
y = k[1]
k[1] = k[0]
k[0] = 5 * y
```

Figure 11: Participant 10 answers this question correctly after iteratively revising an initial guess of a solution.

correct output. They complete the problem by selecting a particular input that meets the derived specification.

> For every element in the list `k` ... we compound the numbers in the list `k` to a maximum of 20. If it hits 20, it stops. ... The max `sum` can be is 20 for it to actually run ... So the max it can have is 20 and the code has to have 2. So, the first two elements of the list, it has to hit that number. So 10 comma 11 comma 5. (writes `[10, 11, 5]`) Is that the right answer?

5.3 How students use `repl.it` on reverse-tracing problems

Students' use of `repl.it` on reverse-tracing problems has some similarity to its use on tracing problems. In particular, there is a lot of usage to check whether an answer is correct. In these cases, the subject will copy the code to `repl.it`, edit it to provide their predicted input value, and then execute the code to see if it produces the desired output. This scenario strongly correlates to the one presented in Section 4.2.1, except that if their answer is incorrect they do not automatically receive the correct answer.

Furthermore, we also observed instances where students used `repl.it` to review aspects of Python semantics or its API as discussed in Section 4.2.2. We consider these two behaviors to be productive because they don't permit trivially solving reverse-tracing problems.

While we saw no direct counterpart to the brute-force execution described in Section 4.2.3, we observed two other behaviors that students attempted to use to correctly answer questions where they could not trace the code. We discuss these next.

5.3.1 Random guessing. Some participants repeatedly executed code on `repl.it` with answers based on seemingly random guesses. In these instances, subjects showed little comprehension of the code. For example, they may try different data types one after another looking for one that doesn't cause syntax errors, or cycle through values of a given data type only checking to see if they produce the desired output. Sometimes the participant tries answers with a variety of distinct characteristics (e.g., even/odd, positive/negative, large/small).

This guessing process is demonstrated by participant 6 successful attempt to complete the question in Figure 10. We discuss common guessing strategies in Section 6.

> Let's start off with trying this and see what happens.
> *(Tries input 0, Output is -1; incorrect, desired solution 2)*
> *(Tries input 1, Output is 0) (Tries input 2, Output is 1)*
> Do that.
> *(Tries input 3, Output is -1)*
> Maybe I should just try doing even numbers.
> *(Tries input 4, Output is -1)*
> No? Try negative numbers that way.
> *(Tries input -2, Output is -1) (Tries input -1, Output is -1) (Tries input -5, Output is -1)*
> Noo, I feel like I am on to something though.
> *(Tries input 10, Output is 4)*
> Oohh lets see.
> *(Tries input 5, Output is -1)*
> So. *(Tries input 6, Output is 2, correct solution)*
> Hold up. Hahahahaha, Nice.

Even when it is possible to obtain credit for doing reverse-tracing problems through random guessing using code execution to check one's guesses, it can be hard to predict how many guesses and, hence, how much time will be required. This is in contrast to tracing problems where students can type and execute the code to obtain the correct answer in a predictable amount of time.

5.3.2 Output-based reasoning. Some participants used repeated `repl.it` executions and educated guessing in an attempt to learn the relationship between the code's input and output—akin to a transfer function—without understanding code itself. Starting with a guess for the input (sometimes the expected output itself), participants use feedback from the computed output to refine their guess towards a correct solution. For example, participant 10 solves a problem shown in Figure 11 by guessing an initial input (the list `[1,2,3]`) then reads the output and iteratively revises their solution until they get the problem correct.

> List equals 1 2 3. I just want to see how this works.
> *(Tries input* `[1,2,3]`*, Output is* `[10,1,3]`*, incorrect from desired solution [5,6,5].)*
> Umm, for it to come out with 5 6 5. Umm, where does the `y` come in? So if 5 is multiplied by this (points at index 0 in initial answer [1,2,3]) so `k[0]` equals 1 and this (`k[1]`) equals that (index 1 of `[1,2,3]`).
> *(Tries input* `[1,1,3]`*, output is* `[5,1,3]`*, incorrect but closer to the desired output [5,6,5].)*
> k[1] is equal 6. I wonder if that changes–
> *(Tries input* `[6,1,3]`*, incorrect output* `[5,6,3]`*, but even closer to desired output* `[5,6,5]`*.)*
> Oh and this would just be–

(Tries input `[6,1,5]`*, gets correct output* `[5,6,5]`*.)*

In another instance, participant 14 traces a computer-based version of the question in Figure 9, except where `limit` is equal to 9. They start tracing the code manually, but seeming uncertain. Then, they copy the code to `repl.it` and try an initial answer of `[3,4,5]` which is incorrect but close to the desired output (incorrect output 3, desired output is 2).

> *(After trying* `[3,4,5]`*)*
> Oh, oh, my god, this has to be greater than 9. Oh, I can't put 10. What am I thinking?
> *(Tries* `[3,4,12]` *on repl.it, leads to same incorrect output 3.)*
> Sum. Sum is what? Oh, wait, it doesn't matter. Uhh 15.
> *(Tries* `[3,15,12]` *on repl.it, leads to desired output 2, gets question correct.)*

In both examples, the code has become secondary to the output computed by `repl.it` as a means for solving the question. In both instances, some knowledge of the code was used to reason out the solution, but neither student seemed likely to correctly answer the question without `repl.it`'s assistance. This is similar to the findings of Bednarikn *et al's* eye-tracking study where expert programmers debugging code related the code to the output systematically [3]. While this behavior is preferable to tracing questions where the answer can be directly computed from the provided code, it would be useful to be able to design questions that mitigate this solution approach.

6 DISCUSSION

While the number of participants is on the low side, as is to be expected in a qualitative study like this, we can see that there is a qualitative difference between tracing and reverse-tracing question in students' ability to trivially identify correct answers on computer-based versions. While reverse-tracing questions are not completely without potential exploits, exploiting them seems to require a larger body of computing knowledge on the part of the student. So from the perspective of identifying a question type that is resilient to computer-based exams, we are cautiously optimistic.

It is less clear, however, that reverse-tracing questions are equivalent pedagogically to tracing questions. Our impression is that the simpler questions—the ones that you might deploy in the first third or first half of the semester—may be sufficiently equivalent pedagogically. As participants were solving these reverse-tracing problems, they showed many problem-solving strategies that appeared similar to tracing strategies found in prior work. For example, in Figure 7, the participant applies two sketching strategies identified by the Leeds Working Group, Computation and Trace [31]. Computation is the process of writing a value based on an arithmetic or compound Boolean expression. Trace is the process of keeping track of variables as they change line-by-line throughout the program. After the participants proposed an input value, they performed the Trace (strategy) consistent with prior sketching studies [13, 14, 31, 43, 47, 52]. Figure 7 also demonstrates the Rewrite strategy identified by Cunningham *et al* [13], where the participant rewrote the code on paper. In general, we see students solving those questions at the granularity of individual statements, and scanning forward and backwards a few lines of code and considering a small number (e.g., ≤ 3) of control flow paths seems to not excessively burden the students that are successful with solving tracing problems on paper.

As the complexity of the reverse-tracing question grows, however, it appears that the skill being tested might diverge from that of complex tracing questions. The exponential growth of control flow paths and possible states of variables quickly becomes too much for novices to maintain in working memory. Instead, those students able to successful solve these harder problems seem to shift to trying to `read` the code and build a mental model of it. In this regard, these complex reverse-tracing problems may end up assessing students more similarly to "Explain in plain English" questions [51] than tracing questions. Exploring this possibility is interesting future work.

Finally, in our observations of students' attempts to outwit our questions, we observed that some of them were easier to exploit than others. We summarize some common guessing patterns and other question pitfalls here, as a service to anyone attempting to write reverse-tracing problems that are potentially less prone to exploits:

(1) Participants had the most success randomly guessing inputs that were small, positive integers (e.g., 1–9). When problems clearly had numerical answers, participants seemed more prone to guessing small, positive integers.
(2) Because reverse-tracing problems may have multiple correct answers, be careful that problems with numerical answers don't have large continuous ranges. One formulation of our problems had the correct answer be any number greater than 40, which meant that any random guess of a large, positive was a correct answer.
(3) Avoid having the answer be a feature of the problem. In one problem that included a list as a given parameter, a participant guessed every value in the list as a possible value for the scalar answer. Similarly, a problem like Figure 8a could be problematic because obvious guesses include the values in the `animals` list and any string not in the list, but we never actually saw a student engage in this strategy.
(4) Avoid questions where there is a piece-wise relationship between the input and the output, like the question shown in Figure 11. These questions permit students to independently test each component of the input to find the correct value for the desired output.

7 LIMITATIONS

Like any study, this work has obvious limitations. First and foremost, we had a relatively small sample size of participants due to the large time commitment to coordinate, run, and analyze each session. In addition, our participants were self selected from the pool that we recruited from. It is likely that our participants were on average among the stronger students in the course, because they had the confidence to participate in a study related to skills learned in the course. As such, we might not have an accurate picture of how the course as a whole would perform on our questions.

Another notable limitation of the study is the presence of the interviewer during the sessions. Even in a proctored exam situation, the proctor would rarely be directly observing a student's work, so our participants likely behaved somewhat differently than they would have on proctored exam, due to social-desirability bias, for example, as discussed in Section 4.2.2.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we analyzed how students used an online IDE, `repl.it`, to solve code tracing (find the output of code) and reverse-tracing (find the input that produces a given output) questions. We found that students were able to brute-force tracing questions using the interpreter, making tracing questions of questionable utility on an exam that permits access to the interpreter. In contrast, we found reverse-tracing questions to be less susceptible to exploitation on computerized exams.

For future work, we believe it would be important to validate these findings using a much larger and more representative sample of students, potentially by including these questions on both paper- and computer-based exams with in the same large enrollment course.

In addition, it would be interesting to explore if the results differ were the study repeated using an interpreter like `PythonTutor` [22] that permits students to single step the code forward and backwards and provides visualizations of the state of the execution. Our observations from these interviews, although not discussed in this paper, were that even with an interpreter like `repl.it`, students couldn't always understand why they had failed to trace code correctly. We hypothesize that the additional insight provided by a tool like PythonTutor could be important in helping students to understand code to assist them in completing even complex reverse-tracing problems.

ACKNOWLEDGMENTS

This work was supported in part by Mohammed Hassan's SURGE and graduate college fellowships at the University of Illinois.

REFERENCES

[1] Joao Paulo Barros. 2018. Students' perceptions of paper-based vs. computer-based testing in an introductory programming course. In *CSEDU 2018-Proceedings of the 10th International Conference on Computer Supported Education*, Vol. 2. SciTePress, 303–308.

[2] Joao Paulo Barros, Luís Estevens, Rui Dias, Rui Pais, and Elisabete Soeiro. 2003. Using lab exams to ensure programming practice in an introductory programming course. In *Proceedings of the 8th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, (ITiCSE)*. 16–20.

[3] Roman Bednarik. 2012. Expertise-dependent visual attention strategies develop over time during debugging with multiple code representations. *International Journal of Human-Computer Studies* 70, 2 (2012), 143–155.

[4] John B. Biggs and K. F. Collis. 1982. *Evaluating the quality of learning : the SOLO taxonomy (structure of the observed learning outcome) / John B. Biggs, Kevin F. Collis.* Academic Press New York. xiii, 245 p. : pages.

[5] Yorah Bosse and Marco Aurélio Gerosa. 2017. Why is programming so difficult to learn? Patterns of Difficulties Related to Programming Learning Mid-Stage. *ACM SIGSOFT Software Engineering Notes* 41, 6 (2017), 1–6.

[6] Yorah Bosse, David F Redmiles, and Marco Gerosa. 2019. Connections and Influences Among Topics of Learning How to Program. In *2019 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–8.

[7] Ruven Brooks. 1983. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies* 18, 6 (1983), 543–554.

[8] Alan C Bugbee Jr. 1996. The equivalence of paper-and-pencil and computer-based testing. *Journal of Research on Computing in Education* 28, 3 (1996), 282–299.

[9] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. Eye movements in code reading: Relaxing the linear order. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 255–265.

[10] Jacabo Carrasquel, Dennis R. Goldenson, and Philip L. Miller. 1985. Competency testing in introductory computer science: the mastery examination at Carnegie-Mellon University. In *SIGCSE '85*.

[11] Jonathan Corley, Ana Stanescu, Lewis Baumstark, and Michael C Orsega. 2020. Paper Or IDE? The Impact of Exam Format on Student Performance in a CS1 Course. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*. 706–712.

[12] Martha E Crosby, Jean Scholtz, and Susan Wiedenbeck. 2002. The Roles Beacons Play in Comprehension for Novice and Expert Programmers.. In *PPIG*. Citeseer, 5.

[13] Kathryn Cunningham, Sarah Blanchard, Barbara Ericson, and Mark Guzdial. 2017. Using tracing and sketching to solve programming problems: replicating and extending an analysis of what students draw. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*. 164–172.

[14] Kathryn Cunningham, Shannon Ke, Mark Guzdial, and Barbara Ericson. 2019. Novice rationales for sketching and tracing, and how they try to avoid it. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*. 37–43.

[15] Adrienne Decker, Lauren Margulieux, and Briana Morrison. 2019. Using the SOLO Taxonomy to Understand Subgoal Labels Effect in CS1. *ICER'19 - Proceedings of the 2019 ACM Conference on International Computing Education Research*, 209–217. https://doi.org/10.1145/3291279.3339405

[16] Ronald F. DeMara, Navid Khoshavi, Steven D. Pyle, John Edison, Richard Hartshorne, Baiyun Chen, and Michael Georgiopoulos. 2016. Redesigning Computer Engineering Gateway Courses Using a Novel Remediation Hierarchy. In *2016 ASEE Annual Conference & Exposition*. ASEE Conferences, New Orleans, Louisiana. https://peer.asee.org/26063.

[17] Françoise Détienne and Elliot Soloway. 1990. An empirically-derived control structure for the process of program understanding. *International Journal of Man-Machine Studies* 33, 3 (1990), 323–342.

[18] K. Anders Ericsson and Herbert A. Simon. 1980. Verbal reports as data. *Psychological Review* 87 (1980), 215 – 251. http://www.library.illinois.edu.proxy2.library.illinois.edu/proxy/go.php?url=http://search.ebscohost.com.proxy2.library.illinois.edu/login.aspx?direct=true&db=hsr&AN=521000032&site=eds-live&scope=site

[19] Sue Fitzgerald, Beth Simon, and Lynda Thomas. 2005. Strategies that students use to trace code: an analysis based in grounded theory. In *Proceedings of the first international workshop on Computing education research*. 69–80.

[20] Scott Grissom, Laurie Murphy, Renée McCauley, and Sue Fitzgerald. 2016. Paper vs. computer-based exams: A study of errors in recursive binary tree algorithms. In *Proceedings of the 47th acm technical symposium on computing science education*. 6–11.

[21] Leo Gugerty and Gary Olson. 1986. Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI conference on human factors in computing systems*. 171–174.

[22] Philip J Guo. 2013. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*. 579–584.

[23] Nanna Suryana Herman, Sazilah Binti Salam, Edi Noersasongko, et al. 2011. A study of tracing and writing performance of novice students in introductory programming. In *International Conference on Software Engineering and Computer Systems*. Springer, 557–570.

[24] Cruz Izu and Claudio Mirolo. 2020. Comparing small programs for equivalence: A code comprehension task for novice programmers. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. 466–472.

[25] Norman Jacobson. 2000. Using On-computer Exams to Ensure Beginning Students' Programming Competency. *SIGCSE Bull.* 32, 4 (Dec. 2000), 53–56. https://doi.org/10.1145/369295.369324

[26] Lisa C. Kaczmarczyk, Elizabeth R. Petrick, J. Philip East, and Geoffrey L. Herman. 2010. Identifying student misconceptions of programming. *SIGCSE: Technical Symposium on Computer Science Education* (2010), 107 – 111. http://www.library.illinois.edu.proxy2.library.illinois.edu/proxy/go.php?url=http://search.ebscohost.com.proxy2.library.illinois.edu/login.aspx?direct=true&db=edb&AN=73688244&site=eds-live&scope=site

[27] Ivar Krumpal. 2013. Determinants of social desirability bias in sensitive surveys: a literature review. *Quality & Quantity* 47, 4 (2013), 2025–2047.

[28] Amruth N Kumar. 2013. A study of the influence of code-tracing problems on code-writing skills. In *Proceedings of the 18th ACM conference on Innovation and technology in computer science education*. 183–188.

[29] Amruth N Kumar. 2015. Solving code-tracing problems and its effect on code-writing skills pertaining to program semantics. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*. 314–319.

[30] Vesa Lappalainen, Antti-Jussi Lakanen, and Harri Högmander. 2017. Towards computer-based exams in CS1. In *CSEDU 2017: Proceedings of the 9th International Conference on Computer Supported Education. Vol. 2, ISBN 978-989-758-240-0*. SCITEPRESS Science And Technology Publications.

[31] Raymond Lister, Elizabeth S Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, et al. 2004. A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin* 36, 4 (2004), 119–150.

[32] Raymond Lister, Colin Fidge, and Donna Teague. 2009. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *Acm sigcse bulletin* 41, 3 (2009), 161–165.

[33] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the fourth international workshop on computing education research*. 101–112.

[34] Claudio Mirolo, Cruz Izu, and Emanuele Scapin. 2020. High-school students' mastery of basic flow-control constructs through the lens of reversibility. In *Proceedings of the 15th Workshop on Primary and Secondary Computing Education*. 1–10.

[35] Motherboard (Tech by Vice). 2021. Schools Are Abandoning Invasive Proctoring Software After Student Backlash. https://www.vice.com/en/article/7k9ag4/schools-are-abandoning-invasive-proctoring-software-after-student-backlash.

[36] Greg L Nelson, Andrew Hu, Benjamin Xie, and Amy J Ko. 2019. Towards validity for a formative assessment for language-specific program tracing skills. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research*. 1–10.

[37] Terence Nip, Elsa L. Gunter, Geoffrey L. Herman, Jason W. Morphew, and Matthew West. 2018. Using a Computer-Based Testing Facility to Improve Student Learning in a Programming Languages and Compilers Course. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (Baltimore, Maryland, USA) *(SIGCSE '18)*. Association for Computing Machinery, New York, NY, USA, 568–573. https://doi.org/10.1145/3159450.3159500

[38] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology* 19, 3 (1987), 295–341.

[39] Anna Agripina Prisacari and Jared Danielson. 2017. Rethinking testing mode: Should I offer my next chemistry test on paper or computer? *Computers & Education* 106 (2017), 1 – 12. https://doi.org/10.1016/j.compedu.2016.11.008

[40] replit. [n.d.]. Instant IDE: Code from your browser. https://replit.com/.

[41] Anni Rytkönen and Venla Virtakoivu. 2019. Comparative student experiences on electronic examining in a programming course-case c. In *Proceedings of the 19th Koli Calling International Conference on Computing Education Research*. 1–10.

[42] Carsten Schulte. 2008. Block Model: an educational model of program comprehension as a tool for a scholarly approach to teaching. In *Proceedings of the Fourth international Workshop on Computing Education Research*. 149–160.

[43] Takayuki Sekiya and Kazunori Yamaguchi. 2013. Tracing quiz set to identify novices' programming misconceptions. In *Proceedings of the 13th Koli Calling International Conference on Computing Education Research*. 87–95.

[44] Ben Stephenson. 2018. An Experience Using On-Computer Programming Questions During Exams. In *Proceedings of the 23rd Western Canadian Conference on Computing Education* (Victoria, BC, Canada) *(WCCCE '18)*. Association for Computing Machinery, New York, NY, USA, Article 11, 6 pages. https://doi.org/10.1145/3209635.3209639

[45] William T Tarimo, Fatima Abu Deeb, and Timothy J Hickey. 2015. Computers in the CS1 Classroom.. In *CSEDU (2)*. 67–74.

[46] Donna Teague, Malcolm Corney, Alireza Ahadi, and Raymond Lister. 2013. A qualitative think aloud study of the early neo-piagetian stages of reasoning in novice programmers. In *Proceedings of the 15th Australasian Computing Education Conference [Conferences in Research and Practice in Information Technology, Volume 136]*. Australian Computer Society, 87–95.

[47] Vesa Vainio and Jorma Sajaniemi. 2007. Factors in novice programmers' poor tracing skills. *ACM SIGCSE Bulletin* 39, 3 (2007), 236–240.

[48] Anne Venables, Grace Tan, and Raymond Lister. 2009. A closer look at tracing, explaining and code writing skills in the novice programmer. In *Proceedings of the fifth international workshop on Computing education research workshop*. 117–128.

[49] Matthew West, Geoffrey L. Herman, and Craig Zilles. 2015. PrairieLearn: Mastery-based Online Problem Solving with Adaptive Scoring and Recommendations Driven by Machine Learning. In *2015 ASEE Annual Conference & Exposition*. ASEE Conferences, Seattle, Washington.

[50] Matthew West, Nathan Walters, Mariana Silva, Timothy Bretl, and Craig Zilles. 2021. Integrating Diverse Learning Tools using the PrairieLearn Platform. In *Seventh SPLICE Workshop at SIGCSE 2021 (Virtual event)*.

[51] Jacqueline Whalley, Raymond Lister, Errol Thompson, Tony Clear, Phil Robbins, P K Ajith Kumar, and Christine Prasad. 2006. An Australasian study of Reading and Comprehension Skills in Novice Programmers, using the Bloom and SOLO Taxonomies. *Eighth Australasian Computing Education Conference (ACE2006)* (2006).

[52] Benjamin Xie, Greg L Nelson, and Andrew J Ko. 2018. An explicit strategy to scaffold novice program tracing. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 344–349.

[53] C. Zilles, R. T. Deloatch, J. Bailey, B. B. Khattar, W. Fagen, C. Heeren, D Mussulman, and M. West. 2015. Computerized Testing: A Vision and Initial Experiences. In *American Society for Engineering Education (ASEE) Annual Conference*.

[54] C. Zilles, M. West, D. Mussulman, and C. Sacris. 2018. Student and Instructor Experiences with a Computer-Based Testing Facility. In *10th annual International Conference on Education and New Learning Technologies (EDULEARN)*.

[55] Craig B Zilles, Matthew West, Geoffrey L Herman, and Timothy Bretl. 2019. Every University Should Have a Computer-Based Testing Facility.. In *CSEDU (1)*. 414–420.