RecurTutor: An Interactive Tutorial for Learning Recursion

SALLY HAMOUDA, Rhode Island College
STEPHEN H. EDWARDS, Virginia Tech
HICHAM G. ELMONGUI, Alexandria University and Umm Al-Qura University
JEREMY V. ERNST and CLIFFORD A. SHAFFER, Virginia Tech

Recursion is one of the most important and hardest topics in lower division computer science courses. As it is an advanced programming skill, the best way to learn it is through targeted practice exercises. But the best practice problems are time consuming to manually grade by an instructor. As a consequence, students historically have completed only a small number of recursion programming exercises as part of their coursework. We present a new way for teaching such programming skills. Students view examples and visualizations, then practice a wide variety of automatically assessed, small-scale programming exercises that address the sub-skills required to learn recursion. The basic recursion tutorial (RecurTutor) teaches material typically encountered in CS2 courses. Students who used RecurTutor had significantly better grades on recursion exam questions than did students who used typical instruction. Students who experienced RecurTutor spent significantly more time on solving recursive programming exercises than students who experienced typical instruction, and came out with a significantly higher confidence level.

CCS Concepts: • **Applied computing** → **E-learning**;

Additional Key Words and Phrases: Recursion, misconceptions, interactive online tutorial, eTextbook

ACM Reference format:
Sally Hamouda, Stephen H. Edwards, Hicham G. Elmongui, Jeremy V. Ernst, and Clifford A. Shaffer. 2018. RecurTutor: An Interactive Tutorial for Learning Recursion. *ACM Trans. Comput. Educ.* 19, 1, Article 1 (November 2018), 24 pages.
https://doi.org/10.1145/3218328

1 INTRODUCTION

Recursion is both one the most important and one of the hardest topics taught in lower division Computer Science courses [7, 22, 26, 48]. While recursion can be viewed as a concept, in practice it is expressed in the form of writing or understanding programs. In this work, we present a new tutorial system for recursion. The proposed approach is based on allowing students to practice a wide

We gratefully acknowledge the support of the National Science Foundation under Grants DUE-1139861, IIS-1258571, and DUE-1432008.
Authors' addresses: S. Hamouda, Department of Mathematics and Computer Science, Rhode Island College, 600 Mount Pleasant Avenue, Prividence, RI, 02908; email: shamouda@ric.edu; S. H. Edwards and C. A. Shaffer, Department of Computer Science, Virginia Tech, Blacksburg, VA 24061; emails: s.edwards@vt.edu, shaffer@vt.edu; H. G. Elmongui, Computer and Systems Engineering, Alexandria University, Alexandria 21544, Egypt; email: elmongui@alexu.edu.eg; J. V. Ernst, Embry-Riddle Aeronautical University, COAS – Worldwide, 600 S. Clyde Morris Blvd., Daytona Beach, FL 32114-3900; email: ernstj1@erau.edu.

1946-6226/2018/11-ART1 $15.00
https://doi.org/10.1145/3218328

variety of automatically assessed, small-scale programming, debugging, and non-programming exercises that address the sub-skills required to learn recursion. Students practice those exercises within the context of a complete tutorial. What makes our approach novel is that we combine the necessary technologies to deliver sufficient and relevant practice to better learn recursion.

Our recursion tutorial is presented on the form of two collections of modules within the OpenDSA eTextbook framework [15, 17], and publicly available through the OpenDSA website at http://opendsa.org. OpenDSA is an open source, community-based project to create a body of materials and infrastructure to generate interactive, eTextbooks for Data Structures and Algorithms courses (DSA) at the undergraduate level. OpenDSA's web-accessible eTextbooks integrate textbook quality text with algorithm visualizations (AVs) and a rich collection of interactive exercises, implemented using HTML5 technology. All exercises are assessed automatically with immediate feedback to the student on whether the exercise was answered correctly. OpenDSA has proved to be successful for learning procedural content such as how a specific algorithm or data structure works [16]. Teaching procedural content in OpenDSA is done through the use of AVs and practice exercises, where students demonstrate their understanding of an algorithm by showing proficiency with the behavior of the given algorithm. In particular, OpenDSA makes heavy use of the concept of a proficiency exercise as pioneered by the TRAKLA2 system [31]. OpenDSA has also proved successful for learning more abstract, analytical material such as basic algorithm analysis topics [13].

Recursion is one of the most difficult of conceptual programming skills, which is why students traditionally have so much trouble with it [38]. Students need to learn conceptual programming skills through techniques that are different from those used to learn procedural content such as the behavior of an algorithm [29]. We want students to move beyond understanding recursive examples that they see, to being able to create their own recursive programs. The best way to learn such skills is through targeted practice exercises [5, 9]. Unfortunately, practice exercises for learning programming skills are time consuming to grade manually. As a consequence, students historically have experienced only a small number of homework and test problems where they actually write recursive functions, and feedback typically comes only long after the student gives an answer. We address these issues by allowing students to work through a collection of small-scale programming exercises, designed to address potential misconceptions, and that are automatically assessed and so provide immediate feedback.

The key contributions of our research are as follows:

(1) A new teaching approach for recursion based on greater student interaction with material that directly addresses student misconceptions that has previously been possible.
(2) An analysis to find the most common misconceptions related to basic recursion.
(3) Exercises that address these misconceptions.
(4) A study to determine how more practice with recursive algorithms addresses student misconceptions on recursion.

2 PRIOR WORK

Most previous research on teaching recursion has focused on abstract discussions of the recursion concept and its control flow [10, 21, 32, 35, 41, 43, 53], comparing recursion to other disciplines [14, 28, 39, 51], new ways to view the concept of recursion [10, 19, 53, 54], and the use of visualization, animation, and games to help students to understand recursion [3, 8, 12, 20, 23, 24, 27, 44–47, 50, 52, 55].

While some of the previous research includes in-class experiments [3, 21, 41, 47, 52], we found only two papers that describe experiments resulting in statistically significant evidence that the proposed teaching methods improve student learning of recursion [47, 49].

Tessler et al. [47] had students play Cargo-Bot to situate learning before they were formally taught recursion. Cargo-Bot is a video game for the Apple iPad in which users direct a robot to move crates to a specified goal configuration. They do this by writing recursive programs in a lightweight visual programming language. The experimental group played Cargo-Bot before receiving a lecture on recursion. The control group received the lecture on recursion and then played Cargo-Bot. Pre-, mid-, and post-tests assessed the students' understanding of recursion in two ways: (1) Students traced a recursive function and determined its return value, and (2) students wrote their own recursive functions to solve a given problem. A *t*-test showed that the learning gains from the pre- to mid-tests for the experimental group are greater than those of the control group for the writing question. However, the Cargo-Bot treatment produced no significant difference in improving students' abilities to trace the execution of recursive functions. The authors did not show the total gain of the pre- to post-test on the two groups, or compare Cargo-Bot against typical instruction.

Tung et al. [49] presented a new approach to teaching recursion, using Visualcode. Visualcode is a visual notation that uses colored expressions and a graphical environment to describe the execution of Scheme programs. Students taking an introductory programming course were divided into two groups of 21 students each. Student's entrance exam scores and prior exam grades were used as control variables. The two students with the highest prior exam grades were assigned to block one, where the two students with the next highest prior exam grades to block two, an so on. The two members of each block were randomly assigned to the experimental units. The experimental group received instructions using Visualcode, while the control group received instructions without using Visualcode. A *t*-test showed that students in the visual group significantly outperformed those who were in the control group in both evaluation questions and programming questions.

Chaffin et al. [3] presented a game that provides computer science students the opportunity to write code and perform interactive visualizations to learn about recursion through depth-first search of a binary tree. They compared the scores on a pre-test and a post-test given to the students before and after using the game. Their analysis showed statistically significant evidence that their proposed teaching methods improve student knowledge of recursion by comparing the pre-test grades to the post-test grades, but did not compare against typical instruction.

We note that none of these studies adopted the pedagogical model of providing an interactive tutorial combined with practice on programming exercises that address basic recursion misconceptions with automated assessment and feedback to the students.

3 REQUIREMENTS GATHERING

This section presents the requirements gathering process for designing the recursion tutorial. First, we present our findings from surveys given to CS instructors regarding their views on how well students are learning recursion using typical instruction methods. Then, we show the results of surveys on the time students actually spend on recursion, and their confidence level when using typical instruction (those students did not use RecurTutor). We study confidence because research shows [11, 33, 42] that confidence is an essential ingredient to valuable engagement and participation in adult learning. It has been observed that students with more confidence were less stressed, more motivated, and acclimatize better to different situations. We also enumerated the different skills required to write and trace a recursive function as determined by previous research [2, 4]. We used the list of required skills when building our basic recursion tutorial.

During our requirements gathering process, we analyzed more than 8,000 student responses to basic recursion programming questions and non-programming (tracing) questions that were written for CS2 exams. From that analysis we found many frequently repeating misconceptions.

Table 1. Instructors Survey Responses on Time for Recursion

Question	Count	Median
Course level	CS2: 11 CS3 : 3	N/A
Background required	None: 12 Basic: 2	N/A
Time on recursion in class	5 to 10h: 12 Unknown: 2	7h
Time required out-of-class	4 to 7h: 2 8 to 27: 12	10h
Students need more time out-of-class	Yes: 14 No: 0	N/A

3.1 Instructor Surveys

Our goals from conducting instructor surveys were (1) to determine if instructors feel that there is a need for better recursion instruction (universally they agree that there is), and (2) to determine the operational parameters that any future educational intervention must operate under in terms of time available for students to study recursion (summary: students ought to be spending more time on recursion out-of-class than historically they have spent).

Participants were instructors who have at least 1 year of experience teaching recursion. We received survey responses from 14 respondents (of 25 contacted) regarding their views on recursion instruction. The instructors belong to six different institutions in three different countries.

The instructor survey questions were as follows:

(1) Briefly describe the course that you are answering this survey for. For example, is it a typical CS1 or CS2 course, or something else?
(2) How much background in recursion do you expect that students will have when they start this course?
(3) Counting actual contact time in the classroom and lab sessions, how much time during the semester do you devote to recursion?
(4) Not counting time spent in a class or lab session, how many hours do you think that the typical student NEEDS to spend on their own to learn and understand the topic of recursion? Include time spent reading the textbook, course notes, or online materials, and the time spent working on homework or practice exercises.
(5) Do you think that the typical student in your course is spending the amount of time necessary to learn and understand recursion?

The results are shown in Table 1. The key findings from the surveys are that instructors spent a median of 7 hours per semester in class covering recursion, and estimated that students need to spend a median of 10 hours learning and practicing recursion outside of class. The median of instructor responses for the number of hours that are needed out of class is 10, with a mean of 11 hours.

3.2 Student Surveys

Our goals from conducting student surveys were to determine (1) the time that students actually spend on recursion and (2) their confidence level as a result of receiving typical instruction on recursion. The participants were students enrolled in CS2114 Software Design and Data Structures during the Spring 2014 semester at Virginia Tech. The students had not used our recursion tutorial, but had been assigned recursion programming exercises from Coding Bat [34].

Table 2. Spring 2014 Students Survey Responses on Time for Recursion

Question	Median
Time on recursion out-of-class	4h
Time on Coding Bat	1.9h
Confidence level	2.5

During the last lab session of CS2114, students were given a paper survey regarding their experience with learning recursion. A total of 54 out of 157 students filled in the survey and returned it to the teaching assistant at the end of the lab. The questions were as follows:

(1) Not counting time spent in class or lab, how many hours have you spent this semester on the topic of recursion? Include time that you spent reading the textbook, course notes, or online materials, and time spent working on homework problems involving recursion.
(2) How many hours did you spend on solving the Coding Bat exercises on recursion?
(3) On a scale of 1–5, rate your confidence level about your mastery of recursion (1 being least confident to 5 being most confident).

The results are shown in Table 2. The key findings from the surveys indicate that the students spent a median total of 4 hours on recursion outside of class, including about 2 hours on solving recursion programming exercises in Coding Bat for homework. This contrasts with the instructors' recommendation to spend 10 hours outside of class.

From the survey results, we confirm that students do not spend nearly the time that instructors estimate to be required out-of-class for practicing recursion. This confirms the instructors' unanimous belief that students were not spending enough time practicing recursion.

3.3 Skills Required to Read and Write Recursive Code

Previous studies split programming skills into tracing and writing [30]. Previous approaches to teaching recursion have asked students to solve both code writing and code tracing problems on tests [3, 21, 41, 47, 52]. However, prior research on teaching recursion has not considered the fact that there are differences between the skills needed for code writing versus code tracing. We address these differences in this section.

We agree with Chi et al. [4] that a successful approach to writing a recursive function comes from thinking in a top-down manner. Successful programmers do not worry about how the recursive call solves the sub-problem. We teach students to simply accept that it will solve it correctly, and to use this result to correctly solve the original problem. For example, if the student is asked to compute $n!$ recursively, she should think in the following way:

—Know that the mathematical function for computing the factorial is $\text{fact}(n) = n * \text{fact}(n-1)$.
—The crucial observation is to not worry about how recursion computes $\text{fact}(n-1)$, simply multiply whatever this is by n.
—Know that the simplest case is $\text{fact}(1) = 1$. The recursive calls stop when n reaches 1.

On the other hand, when it is required to read or trace a recursive function, we agree with Bhuiyan et al. [2] that the most useful approach is to think about it with a stack model. This is traditionally how instructors first present the recursive process. That means that each call to the recursive function can be viewed as the opening of a new box and the prior box is stacked until a base case is reached. The corresponding returns from the function calls are the closures of boxes on a last-in-first-out basis.

It is important to recognize that these two approaches are quite different ways of thinking about recursion. Writing a function is best done by ignoring the details of recursive processing, focusing solely on the result of the recursive call. It is just as if a call was being made to some library function that the programmer has no information about other than its outcome. In contrast, tracing the behavior of an unknown recursive function in order to deduce its behavior requires the opposite mode of thinking, where the details of the process are traced in the same way that the computer would execute the recursive function calls.

3.4 Driving Hypothesis

We hypothesize that difficult programming concepts like recursion are best learned by an approach that involves a lot of practice exercises, and that students will achieve a better understanding of recursion through this approach. Since writing a recursive function involves a different thought process from understanding the behavior of a recursive function, practice in both is needed.

From our initial surveys we have found that the traditional instructional process, as reported by the instructors, failed to get students to spend as much time on recursion as the instructors believed was necessary for proper understanding of the material. So one goal for our tutorial is to force more constructive engagement with the material. The result will be more time spent, but this is a side effect, not the direct goal. The instructor and student survey results set requirements for the basic recursion tutorial's estimates of the time that students will need to spend on recursion out of class (a median of 10 hours) in order to achieve proper learning. To get students to engage more productively, we have them do many practice exercises that are designed to address their misconceptions. The recognition that there are differences in the skills required to write and to understand a recursive function also sets a requirement on how the recursion tutorial should be organized and ordered. Explicit practice is needed with both aspects of the process.

3.5 Identify Basic Recursion Misconceptions

Instructors need to understand subject matter deeply and flexibly so that they can help students create useful mental maps, relate various ideas to each other, and address misconceptions [36]. That is why we took as a prerequisite for building the recursion tutorial to first find common student misconceptions.

To generate our list of misconceptions, we began with reviewing the existing research literature [1, 20, 21, 40, 41]. We then analyzed a large corpus of student answers to exam questions to further refine our list. We analyzed approximately 8,000 responses to recursion questions given to students over three semesters in pre-test, post-test, midterm, or final exams of a traditional CS2 course. Table 3 shows the number of students and the number of recursion questions for each test.

We have chosen to present our findings from the analysis of student answers and research literature as a list of misconceptions and difficulties, inspired by Ragonis and Ben-Ari's work on object-oriented programming [37]. A misconception is a mistaken idea or view resulting from a misunderstanding of something. Difficulty here means the empirically observed inability to do something. It is possible that a student exhibits a difficulty due to an underlying misconception (possibly one already listed here or one so far unidentified). A difficulty might also result because the student lacks some skill or knowledge.

We categorize the difficulties and misconceptions by related topic. We give each an identifying tag, to be used in our analysis presented in Section 5.3. We also indicate our source for each item, whether from the literature or from a type of question in our body of student responses. We note that the literature tends not to give sufficiently precise descriptions of misconceptions or difficulties for this purpose. We originally came up with a longer list than is shown here, but some items

Table 3. Number of Students and Number of Recursion Questions per Exam

Exam Term	Students	Questions
Pre-test Sp14	152	10
Mid-term SP14	160	5
Pre-test F14	178	8
Mid-term F14	216	4
Post-test F14	203	8
Pre-test Sp15	166	5
Mid-term SP15	43	5
Final SP15	167	4

were pruned based on the advice of the instructor surveys or their lack of representation in actual student responses.

Backward Flow.

(1) Misconception: No statements that appear after the recursive call will execute. This misconception was found in student answers to tracing questions that had code after the recursive function. This misconception was also found in [20, 40, 41]. [BFneverExecute]
(2) Misconception: Statements that appear after the recursive call will execute before the recursive call is executed. This misconception was found in student answers to tracing questions that had code after the recursive function. [BFexecuteBefore]

Infinite Recursion.

(1) Misconception: If there is a base case, then it will always execute. If the recursive call does not reduce the problem to the base case, then the base case will return and that will terminate the recursive method. This misconception was found in student answers to tracing questions that had code after the recursive function. [InfiniteExecution]

Recursive Call.

(1) Difficulty: Cannot formulate a recursive call that eventually reaches the base case. This misconception was found in student answers to writing and code completion questions. [RCwrite]
(2) Misconception: A value will be returned from a recursive call even if the `return` keyword is omitted. This misconception was found in student answers to code writing questions. [RCnoReturnRequired]
(3) Misconception: All recursive functions require the `return` keyword (even when the recursive function does not return a value). This misconception was found in student answers to code writing questions. [RCreturnIsRequired]

Base Case.

(1) Misconception: The base case must appear before the recursive call. The base case must be in the `if` condition while the recursive call has to be in the `else` condition or an `if else` condition. (The student therefore has difficulty recognizing whether the recursive call or the base case is executed when tracing code.) This misconception was found in student answers to code tracing questions. [BCbeforeRecursiveCase]

(2) Misconception: The base case action must always return a constant, not a variable. This misconception was found in student answers to code completion questions. [BCactionReturnConstant]
(3) Misconception: The base case condition must always check a variable against a constant, not against another variable. This misconception was found in student answers to code completion questions. [BCcheckAganistConstant]
(4) Difficulty: Cannot write a correct base case. The student is given a description for what a function should do, and an incomplete implementation for the function with a missing or incorrect base case. The student has difficulty coming up with a correct base case to complete the implementation. This misconception was found in student answers to code writing questions. [BCwrite]
(5) Difficulty: Cannot properly evaluate the base case, such that the student believes that the recursive method executes one more or one less time than it should. This misconception was found in student answers to code tracing questions. This misconception was also found in [20, 40]. [BCevaluation]

Updating Variables.

(1) Misconception: Prior to the recursive call, we can (within the recursive function) define a "global" variable that is initialized once and updates when each recursive call is executed. This misconception was found in student answers to code writing questions. [GlobalVariable]

After this phase of identifying misconceptions and difficulties from the literature and from our analysis of the corpus of student responses on exams, we then gathered feedback from a group of 14 instructors regarding their views on the importance of the items on our list, and soliciting other possible misconceptions and difficulties that we might have missed. This process of gathering instructor input was done for two reasons: both to help orient the tutorial material to address the right misconceptions and difficulties, and also as a step in developing a Concept Inventory on recursion [25]. The recursion misconceptions as identified helped us to frame the exercises, visualizations, and prose in the tutorial so as to directly target those misconceptions.

4 RECURTUTOR

From our initial surveys we have found that most instructors acknowledge that their traditional instructional practice did not result in students that are adequately proficient with recursion, and they recognize specifically that students do not spend enough time or get enough practice with recursion. We agree that "practice makes perfect," and that the best known way to learn a skill is to work on many practice problems [5, 9]. Our analysis of the research literature and a large corpus of student answers to exam questions revealed a collection of specific misconceptions that are more-or-less common for students. Thus, the key goal for our tutorial was to increase the amount of active practice that students get, especially practice on a reasonable collection of tracing and programming exercises, with content and exercises designed to explicitly address the identified common misconceptions and difficulties.

A specific issue to be addressed in the tutorial relates to our recognition that there are at least two distinct skills required to learn recursion: writing and tracing. We observe that if a student is asked to write a recursive function, a successful way is to not worry about how the recursive call solves the sub-problem. On the other hand, if it is required to read or trace a recursive function, then the student will need to think in a bottom-up manner. That means the student will evaluate the base case and work backward until reaching the required function call.

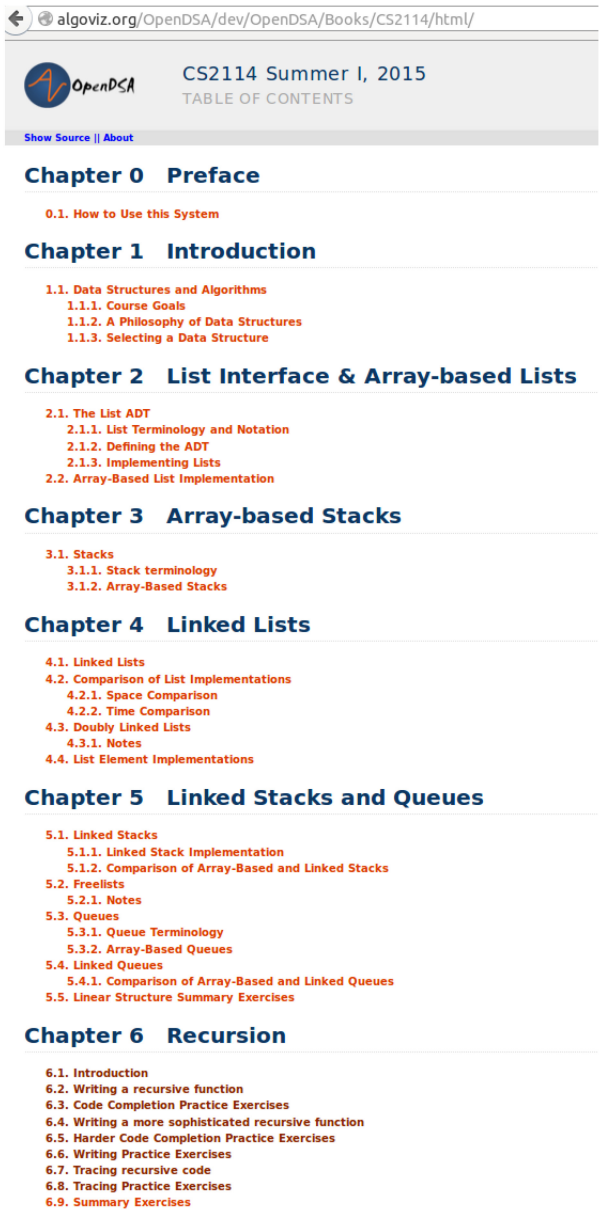

Fig. 1. RecurTutor in the CS2114 eTextbook.

We have developed a recursion tutorial based on these principles, named **RecurTutor**. RecurTutor is presented to users as a chapter (where a chapter is defined as a series of modules) within the OpenDSA eTextbook system. Figure 1 shows RecurTutor as Chapter 6 in the eTextbook for a CS2 course (named CS2114 in the figures). An example module from RecurTutor is shown in Figure 2.

RecurTutor provides automatic assessment for its practice exercises, giving immediate feedback without putting additional grading burden on the instructor. RecurTutor exposes students to

Fig. 2. An example of a lesson (module) in RecurTutor.

interactive exercises that fill in the gap between the amount of practice required by the instructors and the practice that students actually get. The practice exercises address the known student misconceptions and difficulties, while remaining within the time expectations expressed by instructors when surveyed. RecurTutor initially had 21 programming exercises, which we later reduced to 20 because one was reported by students to be unreasonably difficult. Each programming exercise asks the student to complete a given recursive function by writing the base case, recursive call, recursive case, or a combination of these. In some cases the student will write a complete recursive function given only its signature and a functional specification.

Figures 3 and 4 show examples of programming exercises. Feedback for the programming exercises is of three types:

—Correct: When the answer is perfectly correct in that it matches the output from the model answer on the test cases.

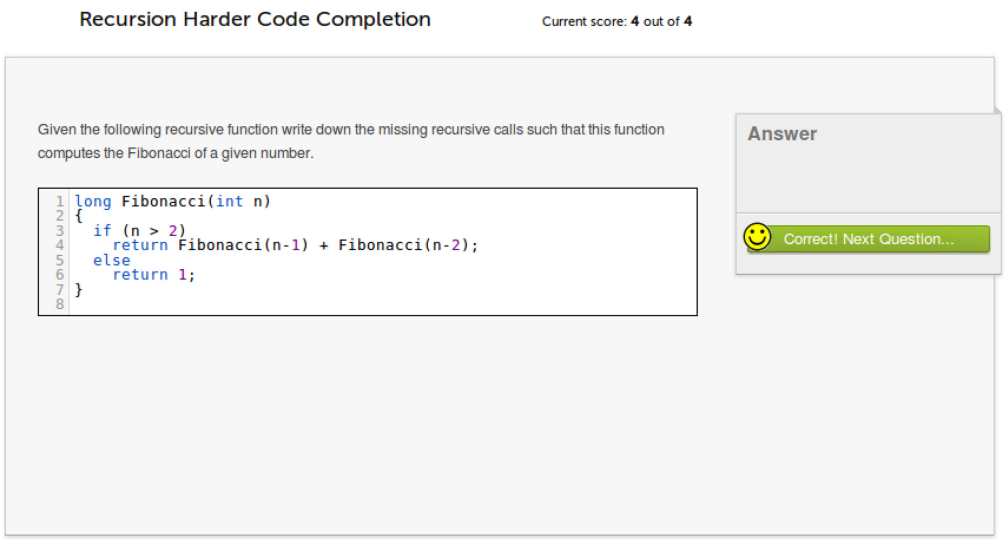

Fig. 3. Code completion programming exercise with feedback indicating the correct answer.

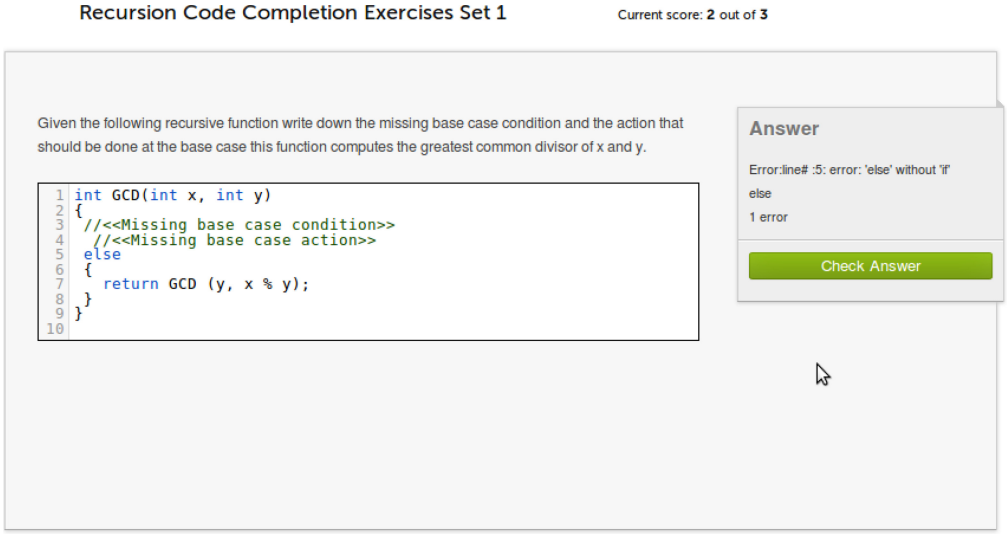

Fig. 4. Code completion programming exercise with feedback on a syntax error.

—Incorrect: When there are no syntax errors but the answer is not correct. The answer may be incorrect because it did not pass the unit tests or because it led to infinite recursion. The feedback message gives information about why the answer is incorrect.
—Syntax error: When there are syntax errors. A full listing of the errors generated by the compiler is shown to the student.

Students can attempt an exercise as many times as they want, and they are given credit when they get a correct solution.

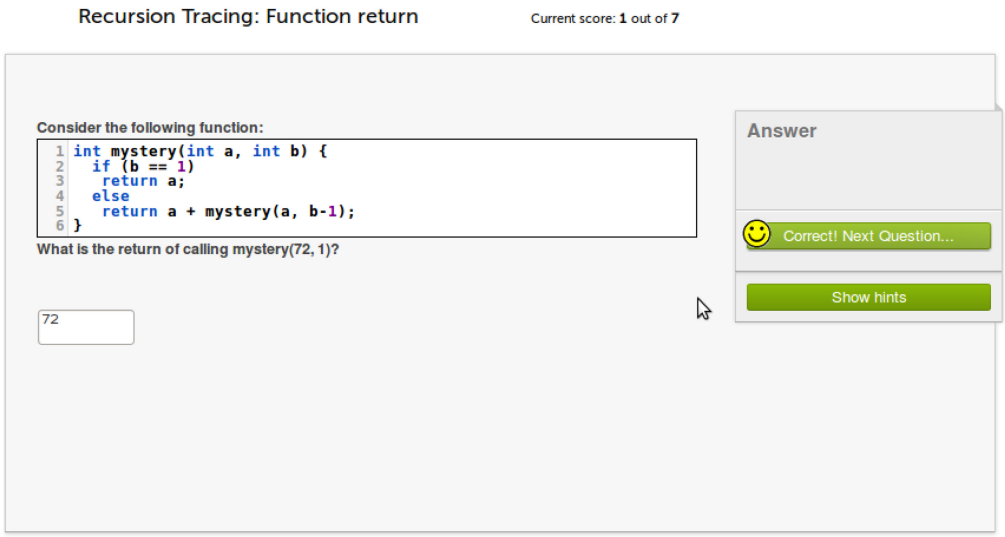

Fig. 5. Fill-in-the-blank style tracing exercise.

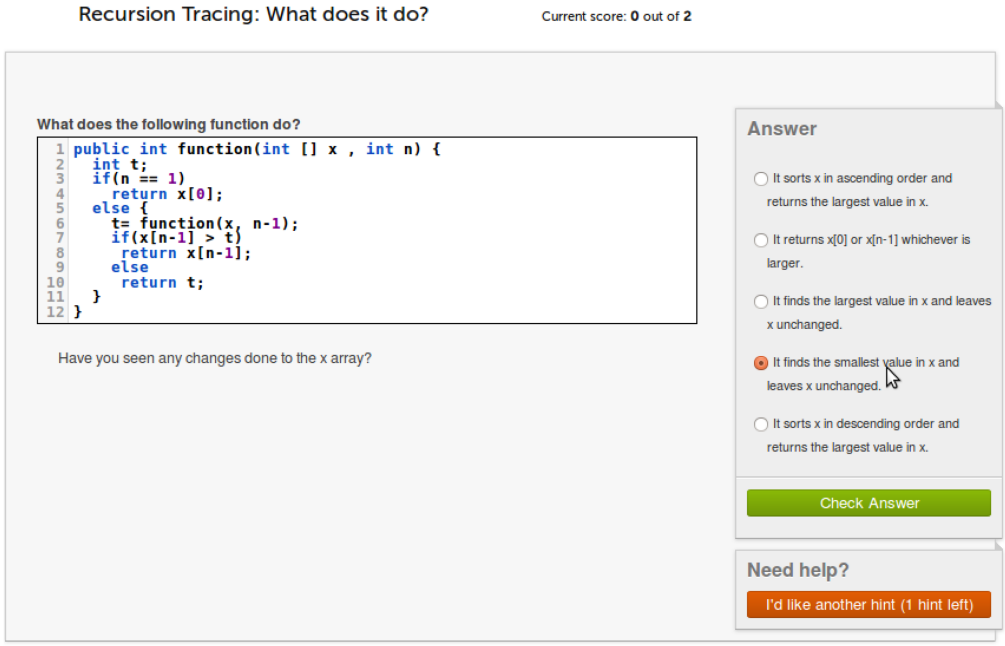

Fig. 6. Multiple choice tracing exercise.

In addition to the 20 programming exercises, there are also 20 tracing exercises that ask students to spot an error in a recursive function, fix an error, or provide the output/return for a given recursive function. Figures 5 and 6 shows example of tracing exercises.

Along with the various exercises, RecurTutor also includes a series of 10 example presentations that are intended to help students overcome the various misconceptions and difficulties. The two slideshows that appear in Figure 2 are examples of such example presentations.

Table 4. Spring 2015 Students Survey Responses on Time on Recursion

Question	Mean
Time on recursion out-of-class	7.3h
Time reading RecurTutor	1.65h
Time on RecurTutor exercises	3.3h
Confidence level	3.06

Descriptions of the various exercises and examples, along with details on how each exercise relates to the items on the misconceptions and difficulty list are presented in [25]. The example presentations in the tutorials explicitly address 6 of the 12 misconceptions and difficulties on our list. There are multiple exercises that explicitly relate to each of the misconceptions or difficulties, except for the two about whether the `return` keyword is required to properly return a value. This misconception is implicitly addressed by many of the programming exercises, as this keyword must be used correctly in order for the recursive function to work.

5 RECURTUTOR'S IMPACT ON STUDENTS

In this section, we present the results from our evaluation of the use of RecurTutor in a CS2 course at Virginia Tech. We compare the outcomes of a control group (sections of the CS2114 course given without using RecurTutor) against an intervention group (sections of the CS2114 course that used RecurTutor). Specifically, we compare the outcomes on the final exams for these sections, and the results from end-of-semester surveys given to the students. Control and experimental groups were taught by the same instructor, and come from the same background with respect to their prerequisite courses. The materials taught in class and labs were the same. Students were not required to have background knowledge on recursion or college-level math before this course.

5.1 Student Confidence and Time Spent Practicing Recursion

In this section, we compare student survey data collected during Spring 2015 (the intervention groups) as compared to student survey data collected during Spring 2014 (the control groups). Comparing those results to the Spring 2015 surveys shows the effect of using RecurTutor on students' confidence level and time spent on recursion. The same survey was given to the control and the intervention group, using the same delivery method.

The participants of the intervention group were students enrolled in CS2114 Data Structures and Software Design during Spring 2015 at Virginia Tech. CS2114 has typical CS2 content, and is a programming-intensive course with 2 hours of programming labs each week. OpenDSA exercises were used as weekly mandatory homework assignments. RecurTutor exercises made up three of those assignments. About 80% of the students voluntarily did additional recursion practice using RecurTutor to prepare for midterms and the final exam.

During the last lab session for CS2114, students were given a paper survey regarding their experience with learning recursion. A total of 83 students completed the survey. Students were not aware prior to its administration that they will be surveyed at the end of the semester. The questions were identical to those used in the 2014 survey, shown in Section 3.2.

To see the effect of RecurTutor, using an unpaired t-test ($\alpha = 0.05$), Spring 2015 survey responses were compared to Spring 2014 survey responses. We verified that the distributions are normal before performing the t-test. The key information from the Spring 2015 survey is shown in Table 4. Table 5 shows the results of the t-test comparing the two survey groups.

Table 5. A *t*-Test Comparing the Time Spent on Recursion for Spring 2015 vs. Spring 2014

	Sp15 (n=83)		Sp14 (n=54)		p-value
	mean	std. dev.	mean	std. dev.	
Time on Recursion (h)	7.3	7.4	4	4.1	0.1385
Time on Prog Ex (h)	3.3	3.4	1.9	1.1	0.0123*
Confidence level	3.06	0.97	2.5	1.09	0.0429*

* = statistically significant.

The findings from the *t*-test can be summarized as follows:

(1) The total time spent on recursion was not significantly increased when RecurTutor is used.
(2) Comparing the time spent on solving CodingBat to the time spent on solving RecurTutor programming exercises, the time spent on RecurTutor exercises is significantly more.
(3) Comparing the confidence level of students who used typical instruction for studying recursion to students who used RecurTutor, the student's confidence level after using RecurTutor is significantly greater.

We checked our interaction log data to verify that the self-reported times on the survey are representative of time that the students actually spent on the tutorial. For the experimental group, we have computed time on task from student logs. The numbers received from the log analysis were reasonably close to the self-reported ones (median of 6.3 hours computed from the log analysis versus 7 hours self-reported). The system used by the control group did not collect log data, so it was hard to independently verify the self-reported times from the control group. But there is no reason to expect that these are less reliable than the self-reported times from the intervention group.

5.2 Exam Scores

We measured the relative performance of the two groups by comparing the post-test (exam) scores for the students who did not use RecurTutor (the control group) versus those who did use it (the intervention group). The intervention group was students enrolled in CS2114 Data Structures and Software Design course during Spring 2015 (n = 168) at Virginia Tech who attended the final exam of the course. We compared the intervention group's final exam scores against students in two sections from Fall 2014 that did not use RecurTutor (the control groups, n = 215 and n = 157).

During both Fall 2014 and Spring 2015, the participants were given the same set of four questions on recursion on the final exam. Below, we have compared the students scores on each recursion question between the following pairs: Spring 2014 versus Spring 2015, Fall 2014 versus Spring 2015, and Spring 2014 versus Fall 2014. The Fall 2014 and Spring 2015 students were given the same four questions on recursion. Spring 2014 students were given only three out of those four questions.

Our first analysis counts each question as being correct or not correct. Tables 6, 7, and 8 show the results of the chi-square analysis comparing the fractions of students who got each question correct for Spring 2014 (control) versus Spring 2015 (intervention), Fall 2014 (control) versus Spring 2015 (intervention), and Spring 2014 (control) versus Fall 2014 (control), respectively. These results are consistent with improved performance by students who used RecurTutor, as will be discussed further. However, the significant difference between the two control groups on the Code Completion question is cause for concern. As will be discussed below, this turned out to be a poor question.

Table 6. Chi-Square for Recursion Questions Comparing Control vs. Intervention Group: Spring 2014 (n = 157) vs. Spring 2015 (n = 168)

Problem	Spring 2014 % correct	Spring 2015 % correct	p-value
Tracing	88.64	98.36	0.0001*
Infinite Recursion	95.45	99.40	0.0001*
Code Completion	70.7	82.74	0.0001*

* = statistically significant.

Table 7. Chi-Square for Recursion Questions Comparing Control vs. Intervention Group: Fall 2014 (n = 215) vs. Spring 2015 (n = 168)

Problem	Fall 2014 % correct	Spring 2015 % correct	p-value
Writing	59.7	69.7	0.0001*
Tracing	93.99	98.36	0.0001*
Infinite Recursion	97.21	99.40	0.019
Code Completion	83.25	82.74	0.5944

* = statistically significant.

Table 8. Chi-Square for Recursion Questions Comparing Two Control Groups: Spring 2014 (n = 157) vs. Fall 2014 (n = 215)

	Spring 2014 % correct	Fall 2014 % correct	p-value
Tracing	88.64	93.99	0.3997
Infinite Recursion	95.45	97.21	0.3367
Code Completion	70.70	83.25	0.005*

* = statistically significant.

While the chi-square test results were good in terms of showing that the intervention has a statistically significant effect, we would like to compute the effect size of the intervention. Unfortunately, this approach does not give us standard deviations to work with. So we reanalyzed the data by assigning correct answers to have a score of 100 and an incorrect answer to have a score of 0. Note that for certain questions, we could also apply partial credit scores (for the chi-square analysis, we had assigned student answers to correct or incorrect based on a score above or below 50%), and this explains the slight differences in the means versus % correct values in the tables.

A t-test was used to compare the RecurTutor group to the control groups. The t-test results were the same as the chi-square analysis in terms of significance. The performance results are shown in Table 9. From the table, we find again that there was a statistically significant improvement in performance for the RecurTutor group on each of the three questions. We note that the code writing question had only been given to one of the two control sections, so for that line of the table, n = 215 instead of n = 367.

We have also computed the effect sizes using Cohen's d formula. For the writing question, the effect size is 0.386, for the tracing question, 0.471, and for the infinite recursion question, 0.253.

Table 9. Control vs. Intervention Group *t*-Test Summary Results

Question	p-value	Effect Size	Control Mean	Intervention Mean
Writing	0.0003*	0.386	59.70	69.70
Tracing	0.0018*	0.471	91.22	98.36
Inf. Rec.	0.0433*	0.253	96.30	99.40

* = statistically significant.

Table 10. Differences between CodingBat Recursion Exercises and RecurTutor Exercises

Factor	CodingBat	RecurTutor
Variety of writing exercises ideas	10 ideas	19 ideas
Types of exercises	Writing	Writing and Tracing
Level of difficulty	Easy to Medium	Easy to Hard
Train students on sub-skills and misconceptions?	No	Yes

These are considered moderate effect sizes. Note that while coercing the dichotomous data into scores for computing *t*-tests gives us values for standard deviation, so that we can calculate effect sizes, doing this exaggerates the standard deviation and so undervalues the actual effect size that we calculate.

The findings from the statistical analysis can be summarized as follows.

(1) The students who used RecurTutor did significantly better on the writing, tracing, and infinite recursion questions than the students who did not.
(2) The students who used RecurTutor did significantly better on the infinite recursion question in one semester than the students who did not. The mean of this question is already over 95%, so it is hard to see improvements. Even on the pre-test, the question's mean score was 91.55%.
(3) The code completion question that was originally used was not a useful question. First, it gave inconsistent results between the two pre-intervention groups studied in Spring 2014 and Fall 2014. It also turned out not to be representative of the type of behavior that students would encounter when writing actual recursive functions, and did not test misconceptions that we had found in the actual body of student responses.
(4) For the code tracing and infinite recursion questions, student scores did not significantly differ between the two control sections when using the typical instruction. As these were both control groups with similar instruction, we do not expect to see a significant difference in scores. We interpret this as support for the hypothesis that using RecurTutor was the reason for the improved scores.

5.3 Treatment Differences between the Control and the Experimental Groups

In this section, we will address the main treatment differences between the control group and the experimental group. Any of these differences or a combination of them could be contributing to the enhancement of student scores, and so differences between them represent potential threats to validity that requires future investigation.

(1) Differences between the CodingBat exercises and RecurTutor exercises: Table 10 shows the main differences between CodingBat recursion exercises solved by the control group and RecurTutor exercises solved by the experimental group. Here, "level of difficulty" is determined from student scores on previous uses of these questions in exams. An "idea"

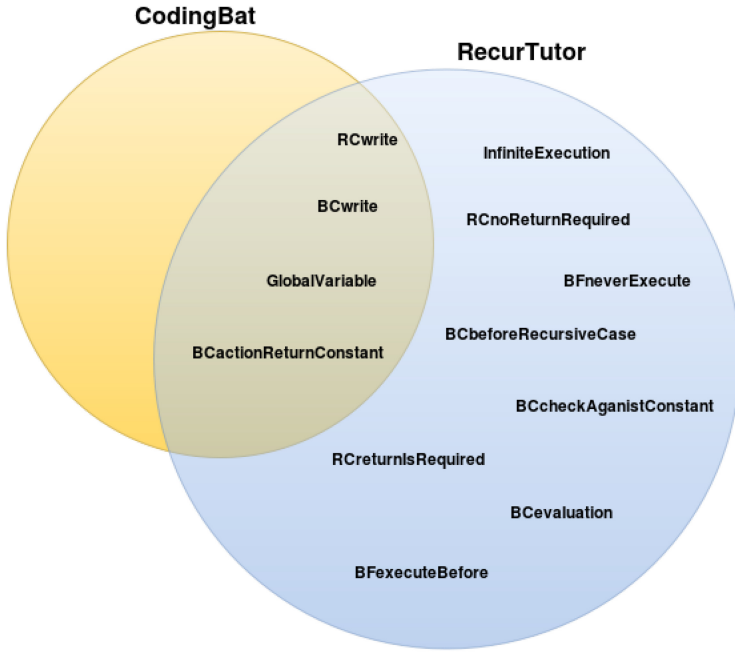

Fig. 7. Misconceptions covered by CodingBat and RecurTutor.

in this table means a requirement. For example, a question that asks the student to write a recursive function that counts the number of a's in a word, has the same idea as an exercise that asks the student to write a recursive function to count the number of c's in a word. Figure 7 shows the misconceptions covered by both CodingBat and RecurTutor. We see from the figure that CodingBat only covers 30% of the misconceptions that we have identified as being encountered by typical students.

(2) The time spent on solving the exercises: The time spent on solving the RecurTutor Exercises was significantly more than the time spent on solving the CodingBat exercises. However, we do not expect that simply spending more time with the CodingBat questions will improve performance, given that CodingBat only covers a subset of the skills necessary for proficiency with recursion.

(3) The style of the questions used on the exams to measure student understanding of recursion: For the writing question used in the exam, we consider it a medium difficulty level question. We believe it does not have a specific style that is more similar to RecurTutor exercises than to CodingBat, or vice versa. For the other two questions, both are considered to be tracing questions, while CodingBat exercises are all writing exercises. So the enhancement in the performance in those questions, although it was not of a big effect size, could be because students were trained on tracing exercises in RecurTutor.

5.4 Exam Questions Item Analysis

We have conducted an item analysis for the exam questions that we used to measure student performance on recursion (on code writing, code tracing, and infinite recursion) [6]. The purpose of doing item analysis is to know if the exam questions that we used can correctly predict student ability on recursion.

Table 11. Difficulty and Discrimination Indices Computed by ltm Package

Question	Difficulty Index	Discrimination Index
Writing	−0.30	1.05
Tracing	−1.63	4.52
Infinite recursion	−1.24	0.96

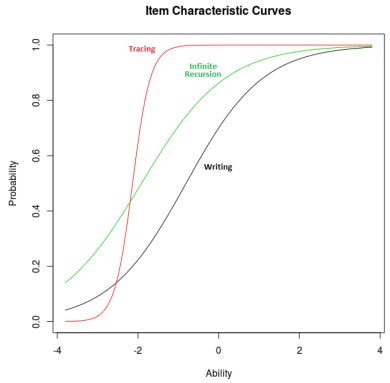

Fig. 8. Item response curves for the questions used to measure student performance on recursion.

We have used the ltm[1] R package to perform the item analysis. We used the two-parameter logistic model, which takes into consideration the discrimination and the difficulty. Table 11 shows the difficulty and discrimination indices computed by ltm.

We mapped the discrimination index computed by ltm to percentages (as computed by Moodle) to be more understandable. The questions all have a discrimination index above 50 (55, 57, and 52, respectively), which is considered as having good ability to discriminate the skill level of the students. The fourth question, on code completion, had a discrimination index below 40, which is considered only fair. It also turns out not to match any of the widely held misconceptions that we have identified on basic recursion. It is also the question that was problematic in the sense that the two control groups had significantly different scores. For these reasons we exclude it from further consideration. The student ability measures are the scores of these three questions, which appeared on the final exam.

We then performed a reliability measure on the remaining three questions together as a test, with a resulting Cronbach $\alpha > 0.9$, which indicates a highly reliable test for the level of knowledge.

As a validity check, we wanted to see if better performance on the recursion questions in the exam correlated with better performance on the individual recursion questions. We evaluated question (item) quality by constructing item response curves (IRCs) using the ltm package. The IRCs for the three exam questions (tracing, writing, and infinite recursion) are shown in Figure 8. The IRC demonstrates the desired correlation between conceptual knowledge and item performance for the three items. So as the student ability increases, the probability of solving the question correctly increases as well. As shown in the IRC, the tracing question is considered to be easier than the other two questions, since students with less ability have a higher probability to get it right than the other two questions.

[1] https://cran.r-project.org/web/packages/ltm/.

Table 12. MANOVA within Quartiles to See If the Number of Tracing Exercises or Writing Exercises Completed Predict Performance on Final Exam Writing Question

Quartile	Prob > F for # of Writing Exs	Prob > F for # of Tracing Exs
A	0.0379*	0.0029*
B	0.0052*	0.0004*
C	0.0058*	<0.0001*
D	0.0049*	<0.0001*

* = statistically significant.

6 PERFORMANCE ON TUTORIAL EXERCISES VS. PERFORMANCE ON EXAMS

In this section, we present findings regarding relationships between student performance on tutorial exercises and later success on recursive writing and tracing questions on the final exam, within each quartile. In particular, we examine the relationships among students of approximately equal performance levels (as defined by being in the same quartile). Within each quartile, we want to see if the number of writing or tracing exercises, or both, solved by the student can predict his or her score on the recursive writing or tracing questions on the final exam, or overall exam scores.

We grouped students into quartiles by sum of scores over all semester exams (and so are believed to be of roughly equal proficiency with the course material, aside from the independent variable). Quartile A represents students with scores above the 75th percentile; Quartile B represents students with scores between the 50th and the 75th percentile; Quartile C represents students with scores between the 25th and the 50th percentile; and Quartile D represents students who performed below the 25th percentile.

We have done a multiple multivariate analysis of variance (MANOVA), where we used the student quartile, the number of writing exercises completed, and the number of tracing exercises completed as the independent variables. The dependent variable is one of the following: the recursion writing question score in the final exam, the recursion tracing question score in the final exam, or the sum of all exam scores over the semester. We are looking to determine if there is a relationship between writing and/or tracing performance on RecurTutor on the one hand, and later success on performance on the other.

Table 12 shows within each quartile how well the number of writing or tracing exercises completed by a student (as a performance measure) can predict their performance in the writing question in the final exam. Table 12 shows that, for each quartile group, the number of writing exercises solved by the student significantly predicts his or her performance on the writing question. It also shows that, for all quartiles, the number of tracing exercises significantly predicts student performance on the writing question. Low performing students had a more statistically significant correlation between the number of tracing exercises solved and the writing question score.

We have repeated the MANOVA, but this time to answer the following question: For students of similar ability level, does performance on tracing exercises or writing exercises predict performance on the final exam tracing question? Table 13 shows that the number of writing exercises solved by the student did not predict student performance on the tracing question in any of the quartiles. Table 13 shows that the number of tracing exercises significantly predicts student performance on the tracing question in all quartiles except for the top quartile. Low performing students who solved more tracing exercises had a better tracing question score in the final exam. The lowest performing student quartile had the smallest p-value.

We then performed MANOVA to answer the question: For students of similar ability level, does performance on tracing exercises or writing exercises predict performance on overall exam score? Table 13 shows that the number of writing exercises solved by the student did not predict

Table 13. MANOVA within Quartiles to See If the Number of Tracing Exercises or Writing Exercises Predict Performance on Final Exam Tracing Question

Quartile	Prob > F for # of Writing Exs	Prob > F for # of Tracing Exs
A	0.341	0.07
B	0.125	0.009*
C	0.048	0.001*
D	0.054	0.0001*

* = statistically significant.

Table 14. MANOVA within Quartiles to See If the Number of Tracing Exercises or Writing Exercises Predict Performance on Overall Exam Scores

Quartile	Prob > F for # of Writing Exs	Prob > F for # of Tracing Exs
A	0.759	0.285
B	0.110	0.668
C	0.552	0.229
D	0.526	0.162

* = statistically significant.

student performance on the overall exam scores in any of the quartiles. Also, the number of tracing exercises did not predict student performance on the overall exam scores in any of the quartiles.

The MANOVA analysis shows that the number of writing and tracing exercises completed by a student can predict his or her score on the recursive writing question on the final exam, and the number of the tracing exercises solved by a student can predict student tracing score in the final exam for students belonging to quartiles B, C, and D but not for quartile A, which has students with the highest performance. That supports our driving hypothesis presented in Section 3.4. We hypothesize that student performance in recursion, measured by the scores on recursion questions, can be enhanced by doing more practice. We can see that the number of tracing exercises completed had the greatest impact on the writing and tracing question scores. The performance on the writing and tracing exercises of the tutorial did not predict the overall exam scores (Table 14). This result gives support to the claim that enhancement of the scores on the writing and tracing questions in the final exam was actually caused by practicing more on the tutorial, rather than related to some other systematic difference among the students in that quartile.

We performed a linear regression analysis to see if the number of tracing and writing exercises solved by the student can predict the overall exam score, or the writing and tracing questions scores, and which quartile has the strongest prediction. We have checked the p-values that tests whether the null hypothesis that the coefficients are equal to 0 for the linear regression between the number of writing exercises solved by student and overall exam scores, the number of writing exercises solved and the writing score, and the number of writing exercises and the tracing question scores. All p-values were significantly low, which means that changes in the predictor are associated with changes in the response variable. In our case, it emphasizes that the number of writing and tracing exercises solved by a student can predict student total of exam scores, final exam recursive writing question score, and final exam recursive tracing question score.

Table 15 shows that the highest R-squared value (determination coefficient) was between the number of tracing exercises solved and the writing question score, which was greater than that between the the number of writing exercises solved and the writing question. We believe this result suggests that the misconceptions covered by the tracing exercises support student writing skills. This still does not explain why practicing writing exercises does not have a greater impact on the

Table 15. R-Square for the Linear Regression Results between the Number of Writing Exercises and Tracing Exercises Solved by Student and Overall Exam Scores, Writing and Tracing Question Scores

Score Type	# of Writing Exs	# of Tracing Exs
Overall exam	0.459	0.26
Writing question	0.38	0.64
Tracing question	0.17	0.52

Table 16. Coefficient for the Linear Regression Model Results between the Number of Writing Exercises and Tracing Exercises Solved by Student and Overall Exam Scores, Writing and Tracing Question Scores

Score Type	# of Writing Exs	# of Tracing Exs
Overall exam	15.23	18.12
Writing question	15.18	17.93
Tracing question	15.25	18.02

writing score than that of the tracing exercises on the writing score. We believe that needs further study.

Table 15 showed that we have low to medium R-squared values. Adding more variables to our model may enhance the R-square values but the data may then contain an inherently higher amount of unexplained variability. For example, many psychology studies have R-squared values less than 50% because people are fairly unpredictable [18]. In our case, we may add additional predictors like number of attempts, the time spent on the exercise, or the time spent on the whole recursion tutorial to our model and see if that will increase the true explanatory power of the model. Our ultimate goal is to know if solving more practice exercises is the cause of better scores. Table 16 shows the linear regression coefficients. In this case, both the number of writing exercises solved and the number of tracing exercises solved appear to relate to outcomes, with the number of tracing exercises having a slightly greater effect.

7 CONCLUSIONS AND FUTURE WORK

Our results support the driving hypothesis presented in Section 3.4, that student performance in recursion can be enhanced by doing more practice that addresses common recursion misconceptions. We have shown that students who used RecurTutor did better than the students who did not use it. The MANOVA analysis showed that the number of tracing and writing exercises solved by a student can predict their scores on the final exam recursive writing and tracing questions, but cannot predict student performance on overall exam scores. The ability to predict performance on recursion-related questions supports the hypothesis that the cause for better enhancement in student scores comes from doing more practice that addresses common recursion misconceptions. Meanwhile, among otherwise similar-performing students, the fact that the amount of practice done on recursion exercises does not predict overall performance (as is expected) serves as an appropriate negative control that the positive correlations between intervention and performance really are related to RecurTutor.

We conclude that the best way to use RecurTutor to enhance student performance on recursion is to allow students to practice recursion by solving the tutorial exercises. However, further analysis is needed to understand what aspects of the practice exercises on RecurTutor leads to the

enhancement in student performance. There are two distinct aspects of RecurTutor that might be affecting the learning of recursion. One is that RecurTutor explicitly delivers instruction aimed at teaching recursion and overcoming misconceptions. The other is that RecurTutor involves extensive practice of recursive skills, both with tracing and with writing recursive functions. Our design is unable to distinguish the relationships between these effects, and this will need to be part of a future study.

As a side effect of its design and delivery, students did spend more time with RecurTutor than previous students reported spending with traditional instruction formats. Instructors perceived that adequate learning required more time than students were spending with traditional methods. This is important in that instructional interventions might often be rejected by instructors if they view them as improving one aspect of instruction at the expense of other topics due to a limited time budget for the students. In this case, the instructors indicated that they felt it appropriate for students to spend additional time. So, it is a positive result in this case that students were willing to actually spend that additional necessary time with RecurTutor.

We have seen that students who solved more tracing exercises did better on the writing question on the exam, while students who solved more writing exercises did not do better on the writing question on the exam. We did not see an effect from doing more writing exercises on tracing exam questions, but that could be because almost all the students already did well on the tracing exercises. We need to further study the relationship between tracing practice and writing practice on the various sub-skills of writing and tracing recursive functions.

We believe that learning hard programming skills is an important area of research that has not been addressed well yet. There are many research ideas that can lead to a better understanding of student misconceptions, where those misconceptions come from, what are the best ways to address those misconceptions, and what are the best ways to measure student understanding of those hard programming skills.

REFERENCES

[1] A. C. Benander and B. A. Benander. 2008. Student monks—Teaching recursion in an IS or CS programming course using the Towers of Hanoi. *Journal of Information Systems Education* 19, 4 (2008), 455–467.

[2] S. Bhuiyan, J. Greer, and G. McCalla. 1990. Mental models of recursion and their use in the SCENT programming advisor. *Knowledge Based Computer Systems*. Springer, 133–144.

[3] Amanda Chaffin, Katelyn Doran, Drew Hicks, and Tiffany Barnes. 2009. Experimental evaluation of teaching recursion in a video game. In *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games (Sandbox'09)*. 79–86.

[4] M. T. H. Chi, R. Glaser, and M. Farr. 2014. *The Nature of Expertise*. Psychology Press.

[5] Ruth C. Clark and Richard E. Mayer. 2011. *E-learning and the Science of Instruction: Proven Guidelines for Consumers and Designers of Multimedia Learning*. Wiley.com.

[6] Linda Crocker and James Algina. 1986. *Introduction to Classical and Modern Test Theory*. ERIC.

[7] Nell B. Dale. 2006. Most difficult topics in CS1: Results of an online survey of educators. *SIGCSE Bulletin* 38, 2 (June 2006), 49–53.

[8] Wanda Dann, Stephen Cooper, and Randy Pausch. 2001. Using visualization to teach novices recursion. *SIGCSE Bulletin* 33, 3 (June 2001), 109–112.

[9] Edward Dillon, Monica Anderson, and Marcus Brown. 2012. Comparing feature assistance between programming environments and their effect on novice programmers. *Journal of Computing Sciences in Colleges* 27, 5 (2012), 69–77.

[10] Jeffrey Edgington. 2007. Teaching and viewing recursion as delegation. *Journal of Computing Sciences in the Colleges* 23, 1 (Oct. 2007), 241–246.

[11] J. Eldred, J. Ward, K. Snowden, and Y. Dutton. 2006. The nature and role of confidence—Ways of developing and recording changes in the learning context. *Adults Learning Journal* (2006).

[12] J. Eskola and Jorma Tarhio. 2002. On visualization of recursion with excel. In *Proceedings of the 2nd Program Visualization Workshop*, Mordechai Ben-Ari (Ed.). HorstrupCentret, Denmark, 45–51.

[13] Mohammed F. Farghally, Kyu Han Koh, Hossameldin Shahin, and Clifford A. Shaffer. 2017. Evaluating the effectiveness of algorithm analysis visualizations. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE'17)*. 201–206.

[14] Gary Ford. 1982. A framework for teaching recursion. *SIGCSE Bulletin* 14, 2 (June 1982), 32–39.
[15] E. Fouh, D. A. Breakiron, S. Hamouda, M. F. Farghally, and C. A. Shaffer. 2014. Exploring students learning behavior with an interactive eTextbook in computer science courses. *Computers in Human Behavior* (Dec. 2014), 478–485.
[16] E. Fouh, S. Hamouda, M. F. Farghally, and C. A. Shaffer. 2016. Automating learner feedback in an eTextbook for data structures and algorithms courses. In *Challenges in ICT Education: Formative Assessment, Learning Data Analytics and Gamification*, S. Caballe and R. Clariso (Eds.). Elsevier Science.
[17] Eric Fouh, Ville Karavirta, Daniel A. Breakiron, Sally Hamouda, Simin Hall, Thomas L. Naps, and Clifford A. Shaffer. 2014. Design and architecture of an interactive eTextbook—The OpenDSA system. *Science of Computer Programming* 88 (2014), 22–40.
[18] Jim Frost. 2014. How to Interpret a Regression Model with Low R-squared and Low P values. Retrieved on September 12, 2018 from http://blog.minitab.com/blog/adventures-in-statistics/how-to-interpret-a-regression-model-with-low-r-squared-and-low-p-values.
[19] Timothy S. Gegg-Harrison. 1999. Exploiting program schemata to teach recursive programming. *Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study, Ablex*, P. Brna, B. duBoulay, and H. Pain (Eds.). 347–379.
[20] Carlisle E. George. 2000. EROSI-visualising recursion and discovering new errors. In *Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education (SIGCSE'00)*. 305–309.
[21] David Ginat and Eyal Shifroni. 1999. Teaching recursion in a procedural environment—How much should we emphasize the computing model? In *The Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'99)*. 127–131.
[22] Ken Goldman, Paul Gross, Cinda Heeren, Geoffrey L. Herman, Lisa Kaczmarczyk, Michael C. Loui, and Craig Zilles. 2010. Setting the scope of concept inventories for introductory computing subjects. *Transactions on Computing Education* 10, 2 (June 2010), Article 5, 29 pages.
[23] Aaron Gordon. 2006. Teaching recursion using recursively-generated geometric designs. *Journal of Computing Sciences in Colleges* 22, 1 (Oct. 2006), 124–130.
[24] Katherine Gunion, Todd Milford, and Ulrike Stege. 2009. Curing recursion aversion. *SIGCSE Bulletin* 41, 3 (July 2009), 124–128.
[25] Sally Hamouda. 2015. *Learning Hard Programming Skills*. Ph.D. Dissertation. Virginia Polytechnic Institute and State University, Blacksburg, Virginia.
[26] Matthew Hertz and Sarah Michele Ford. 2013. Investigating factors of student learning in introductory courses. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE'13)*. 195–200.
[27] Wen-Jung Hsin. 2008. Teaching recursion using recursion graphs. *Journal of Computing Sciences in Colleges* 23, 4 (April 2008), 217–222.
[28] Robert L. Kruse. 1982. On teaching recursion. In *Proceedings of the 13th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'82)*. 92–96.
[29] D. Midian Kurland, Roy D. Pea, Catherine Clement, and Ronald Mawby. 1986. A study of the development of programming ability and thinking skills in high school students. *Journal of Educational Computing Research* 2, 4 (1986), 429–458.
[30] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. 2004. A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bulletin* 36, 4 (June 2004), 119–150.
[31] L. Malmi, V. Karavirta, A. Korhonen, J. Nikander, O. Seppälä, and P. Silvasti. 2004. Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. *Informatics in Education* 3, 2 (Sept. 2004), 267–288.
[32] Claudio Mirolo. 2010. Learning (through) recursion: A multidimensional analysis of the competences achieved by CS1 students. In *Proceedings of the 15th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'10)*. 160–164.
[33] M. Norman and T. Hyland. 2003. The role of confidence in lifelong learning. *Educational Studies* 29, 2-3 (2003), 261–272.
[34] N. Parlante. 2011. Codingbat: Code practice. Retrieved on September 12, 2018 from http://codingbat.com.
[35] Irene Polycarpou, Ana Pasztor, and Malek Adjouadi. 2008. A conceptual approach to teaching induction for computer science. *SIGCSE Bulletin* 40, 1 (March 2008), 9–13.
[36] Barbara Z. Presseisen. 2008. *Teaching for intelligence*. Corwin Press.
[37] Noa Ragonis and Mordechai Ben-Ari. 2005. A long-term investigation of the comprehension of OOP concepts by novices. *Computer Science Education* 15, 3 (2005), 203–221.
[38] Anthony Robins, Janet Rountree, and Nathan Rountree. 2003. Learning and teaching programming: A review and discussion. *Computer Science Education* 13, 2 (2003), 137–172.
[39] Manuel Rubio-Sánchez and Isidoro Hernán-Losada. 2007. Exploring recursion with Fibonacci numbers. *SIGCSE Bulletin* 39, 3 (June 2007), 359–359.

[40] Ian Sanders and Tamarisk Scholtz. 2012. First year students' understanding of the flow of control in recursive algorithms. *African Journal of Research in Mathematics, Science and Technology Education* 16, 3 (2012), 348–362. Retrieved on September 12, 2018 from http://www.tandfonline.com/doi/pdf/10.1080/10288457.2012.10740750.

[41] Tamarisk Lurlyn Scholtz and Ian Sanders. 2010. Mental models of recursion: Investigating students' understanding of recursion. In *Proceedings of the 15th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'10)*. 103–107.

[42] T. Schuller, A. Brassett-Grundy, A. Green, C. Hammond, and J. Preston. 2002. *Learning, Continuity and Change in Adult Life. Wider Benefits of Learning Research Report*. ERIC.

[43] Raja Sooriamurthi. 2001. Problems in comprehending recursion and suggested solutions. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE'01)*. 25–28.

[44] John Stasko, Albert Badre, and Clayton Lewis. 1993. Do algorithm animations assist learning: An empirical study and analysis. In *Proceedings of the INTERCHI'93 Conference on Human Factors in Computing Systems (INTERCHI'93)*. 61–66.

[45] Ben Stephenson. 2009. Using graphical examples to motivate the study of recursion. *Journal of Computing Sciences in Colleges* 25, 1 (Oct. 2009), 42–50.

[46] Linda Stern and Lee Naish. 2002. Visual representations for recursive algorithms. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE'02)*. 196–200.

[47] Joe Tessler, Bradley Beth, and Calvin Lin. 2013. Using Cargo-Bot to provide contextualized learning of recursion. In *Proceedings of the 9th Annual International ACM Conference on International Computing Education Research (ICER'13)*. 161–168.

[48] Allison Elliott Tew and Mark Guzdial. 2011. The FCS1: A language independent assessment of CS1 knowledge. In *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE'11)*. 111–116.

[49] Sho-Huan Tung, Ching-Tao Chang, Wing-Kwong Wong, and Jihn-Chang Jehng. 2001. Visual representations for recursion. *International Journal of Human-Computer Studies* 54, 3 (March 2001), 285–300.

[50] J. Angel Velazquez-Iturbide, Antonio Perez-Carrasco, and Jaime Urquiza-Fuentes. 2008. SRec: An animation system of recursion for algorithm courses. *SIGCSE Bulletin* 40, 3 (June 2008), 225–229.

[51] Susan Wiedenbeck. 1988. Learning recursion as a concept and as a programming technique. *SIGCSE Bulletin* 20, 1 (Feb. 1988), 275–278.

[52] Derek Wilcocks and Ian Sanders. 1994. Animating recursion as an aid to instruction. *Computers and Education* 23, 3 (1994), 221–226.

[53] Michael Wirth. 2008. Introducing recursion by parking cars. *SIGCSE Bulletin* 40, 4 (Nov. 2008), 52–55.

[54] Cheng-Chih Wu, Nell B. Dale, and Lowell J. Bethel. 1998. Conceptual models and cognitive learning styles in teaching recursion. *SIGCSE Bulletin* 30, 1 (March 1998), 292–296.

[55] Chen-Chih Wu, Greg C. Lee, and Janet Mei-Chuen Lin. 1998. Visualizing programming in recursion and linked lists. In *Proceedings of the 3rd Australasian Conference on Computer Science Education (ACSE'98)*. 180–186.

Received July 2016; revised October 2017; accepted February 2018