Enhancing Learning of Recursion

Sally Mohamed Fathy Mo Hamouda

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Applications

November 09, 2015
Blacksburg, Virginia

Keywords: Recursion, Online Learning, Automated Assessment, Concept Inventory, Binary Trees

Enhancing Learning of Recursion

Sally Mohamed Fathy Mo Hamouda

(ABSTRACT)

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 hard to grade. As a consequence, students experience only a small number of problems. The dearth of feedback to students regarding whether they understand the material compounds the difficulty of teaching and learning CS2 topics.

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. The advanced recursion in binary trees tutorial (BTRecurTutor) covers advanced recursion techniques most often encountered post CS2. It provides detailed feedback on the students' programming exercise answers by performing semantic code analysis on the student's code.

Experiments showed that RecurTutor supports recursion learning for CS2 level students. Students who used RecurTutor had statistically significant better grades on recursion exam questions than did students who used a typical instruction. Students who experienced RecurTutor spent statistically significant more time on solving programming exercises than students who experienced typical instruction, and came out with a statistically significant higher confidence level.

As a part of our effort in enhancing recursion learning, we have analyzed about 8000 CS2 exam responses on basic recursion questions. From those we discovered a collection of frequently repeated misconceptions, which allowed us to create a draft concept inventory that can be used to measure student's learning of basic recursion skills. We analyzed about 600 binary tree recursion programming exercises from CS3 exam responses. From these we found frequently recurring misconceptions.

The main goal of this work is to enhance the learning of recursion. On one side, the recursion tutorials aim to enhance student learning of this topic through addressing the main misconceptions and allow students to do enough practice. On the other side, the recursion concept inventory assesses independently student learning of recursion regardless of the instructional methods.

This work received support from the VT-MENA program of Egypt, and the CS National Science Foundation under Grant Numbers DUE-0836940, DUE-0937863, and DUE-0840719.

Dedication

This dissertation is dedicated to the loving memory of my mother, Nadia. Her support, encouragement, and constant love have sustained me throughout my life.

I also dedicate this dissertation to my husband, Mohamed. I give my deepest expression of love and appreciation for the encouragement that he gave and the sacrifices he made during my graduate program.

Acknowledgments

First of all, all thanks due to ALLAH. May His peace and blessings be upon his prophet for granting me the chance to successfully complete my PhD.

My heartfelt gratitude to my advisor, Professor Clifford A. Shaffer for his inspiration, enthusiasm, invaluable guidance, and patience. He has taught me many things, and this work would not have been possible without his encouragement and support. I would like to especially thank Dr. Stephen Edwards for his precious feedback and guidance. I would also like to thank my other committee members Dr. Hicham El Mongui, Dr. Jeremy Ernst and Dr. Aditya Prakash for their support, feedback and efforts reviewing my work and dissertation. It is my honor to have worked and learned from them.

I would like to thank the OpenDSA research group Eric Fouh, and Mohamed Farghally who helped me through out my work. I would like to extend my thanks to all the instructors who used OpenDSA in their classes, and to all the students who used OpenDSA.

I wish to express my sincere gratitude to VT-MENA program director, Prof. Sedki Riad for his endless support and invaluable guidance. He has been and will always be a real father for us.

Special thanks to my mum who inspired me and supported me with her love and blessings throughout my life. She has been a constant source of inspiration, motivation, and strength. I will be ever grateful for her assistance, and am sorry that she did not live to see me graduate. I thank Mohamed, my husband, for his unconditional support all through this journey. I thank Ahmed and Abdulrahman, my children, for being so patient with a busy mum. I thank my mother-in-law for her support and encouragement.

I also thank all my wonderful friends in Blacksburg, Randa, Doaa, Sherin, and Eman, who supported me during my hard times. Thanks to the people in the CSA department. Lastly, I thank Virginia Tech and the Blacksburg community for making my stay here a memorable one.

Once again, thanks to ALLAH All mighty God for giving me the ability, mindset, and perseverance to be where I am now.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Recursion is one the most important and hardest topics in lower division computer science courses [36, 12, 106, 107]. While recursion can be viewed as concept, in practice it is expressed in the form of writing or understanding programs Thus, the best way to learn it is through targeted practice exercises [10]. The best practice problems for such courses are traditionally time consuming to grade. As a consequence, students normally experience only a small number of homework and test problems, whose results come only long after the student gives an answer. The dearth of feedback to students regarding whether they understand the material compounds the difficulty of teaching and learning CS2 topics [25].

In this work, we present a new way for teaching such programming skills. The proposed approach is based on allowing students to practice a wide 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 in order to deliver sufficient and relevant practice to better learn the material. The basic recursion tutorial (RecurTutor) allows students to practice a large number of exercises. The recursion in binary trees tutorial (BTRecurTutor) provides detailed feedback on students' misconceptions detected in the programming exercise answers by performing semantic code analysis on the student's code.

We have created two tutorials, presented on the form of two collections of modules within the OpenDSA eTextbook framework. OpenDSA is an open source, community-based effort to create a complete active-eBook for Data Structures and Algorithms courses (DSA) at the undergraduate level. OpenDSA defines active-eBooks as going beyond classic hypertextbooks, being a close integration of text and images with interactive visualizations/simulations and assessment activities [96]. OpenDSA integrates textbook quality text with algorithm visualizations (AVs) and a rich collection of interactive exercises available as a series of tutorials implemented using HTML5 technology. OpenDSA modules combine content materials in the form of text, slideshow, simulation, and various types of assessment questions. 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. Teaching this procedural content in OpenDSA is done through the use of AVs and exercises. However, conceptual programming skills are different from procedural knowledge of an algorithm. Recursion can be viewed as an example of conceptual programming skills. It is among the most difficult of conceptual programming skills, which is why students traditionally have so much trouble with it. Students need to learn conceptual programming skills through techniques that are different from those used to learn procedural content. We need students to move beyond understanding examples to being able to create their own programs. This requires some sort of internal synthesis of understanding, which is difficult to achieve. The fundamental goal of our work is to develop and evaluate effective means for learning such conceptual programming skills.

As a part of our effort in enhancing recursion learning, we have analyzed more than 8000 basic recursion programming and non-programming question student responses that were written for CS2 exams. From that analysis we found many frequently repeating misconceptions.This in turn lead us to create a draft concept inventory that can be used to measure student's understanding of basic recursion. We have also analyzed more than 600 student responses to questions on recursion in binary tree that were written for CS3[*] exams. From that we have learned many frequently repeating misconceptions, which lead us to create a draft concept inventory that can be used to measure student's learning of recursion in binary tree skills.

The main goal of this work is to enhance the learning of recursion. On one side, the recursion tutorial aims to enhance student learning of this topic through addressing the main misconceptions and through allowing student to do enough practice. On the other side, the recursion concept inventory assesses independently student learning of recursion regardless of the instruction used in teaching.

1.1 Research questions and contributions

Our main research questions are:

1. **Does the amount of time that students spend in typical classes learning and practicing recursion match the amount of time that instructors believe to be required to learn and understand this topic?**
2. **Did using the basic recursion tutorial enhances the confidence level of the students on basic recursion?**
3. **Did using our tutorial lead to students spending time in line with what**

[*]In this dissertation we used the term "CS3" to open refer to a Data Structures and Algorithms course taken subsequent to a traditional CS2 course

instructors believe to be appropriate for learning and understanding basic recursion?

4. **Did using the basic recursion tutorial support basic recursion learning than the typical instruction?**
5. **Does the basic recursion draft concept inventory cover student misconceptions about basic recursion?**

The key contributions of our research are:

1. A new teaching approach for recursion based on greater student interaction with the material than has previously been possible.
2. A study to determine the effect of more practice on learning recursion.
3. Exercises that address the main common misconceptions students have in learning recursion.
4. Infrastructure to support automatic assessment for programming exercises, which will help by giving the students immediate feedback without putting additional grading burden on the instructor.
5. Infrastructure to support semantic code analysis of students answers for programming exercises, which gives students a detailed feedback on their misconceptions.
6. A draft concept inventory for measuring student understanding of basic recursion skills.
7. An analysis to find out the most common misconceptions in understanding recursion in binary trees.

1.2 Key Experimental Results

Experiments showed that RecurTutor had a positive impact on CS2 level students. Students who used RecurTutor had statistically significant better grades in the recursion final exam questions than the students who experienced typical instruction on recursion. Students who used RecurTutor spent statistically significant more time on solving programming exercises than students who experienced typical instruction, and also came out with a statistically significant higher confidence level.

We did an initial administration for the initial draft basic recursion concept inventory. We learned that it include some weak questions that can be excluded in the next version.

1.3 Dissertation organization

We present a literature review in Chapter 2. Chapter 3 presents the requirements gathering for building the basic recursion tutorial and the concept inventory. Chapter 4 describes the basic recursion tutorial (RecurTutor) content, visualizations and exercises that addresses the basic recursion skills. Chapter 5 describes evaluation regrading effect of using RecurTutor on student performance and non-performance outcomes. Chapter 6 describes our efforts to build a draft concept inventory for basic recursion. In Chapter 7, we describe our work on finding student misconceptions regarding recursion on binary trees, and building a tutorial to address those misconceptions (BTRecurTutor). Finally, Chapter 8 presents a brief summary of the tasks accomplished so far, along with future research directions.

Chapter 2

Related Work

In this chapter we present prior research related to different aspects of our work. We start with related work on e-Textbooks. Then we describe related work on teaching recursion, automated assessment, and concept inventories.

2.1 e-Textbooks

This section presents recent work on e-Textbooks that include automatically assessed exercises. At this time, there is much commercial interest in those systems. Some commercial online tutoring systems are so new that their impact has yet to be felt, such as Zyante and recent efforts by Intel and Microsoft.

TRAKLA2 [63, 56] comprises a large collection of Algorithm Visualization simulation exercises with automatic feedback. TRAKLA2 exercises are called "visual algorithm simulations"because they ask students to determine a series of operations that will change the state of the given data structure to achieve some outcome. For example, students might build a tree data structure by repeatedly dragging new values to the correct locations in the tree. Alternatively, the student can gain understanding by examining a step-by-step execution of the algorithm (called the model solution). Many TRAKLA2 exercises include some tutorial text along with pseudocode to explain the algorithm, but their main purpose is to provide an interactive proficiency exercise. A database stores user information, including submissions and grades, as well as information about courses and exercises such as deadlines and maximum points. TRAKLA2 has no exercises to teach recursion or pointer-manipulation.

OpenDSA makes heavy use of the TRAKLA2 visual algorithm simulation concept for its "proficiency exercises", and the TRAKLA2 developers are part of the OpenDSA development team. It also makes use of Khan Academy infrastructure [54] to implement various exercises.

CodingBat [75] is a free site with coding problems used to build coding skill in Java, and now in Python. The problems could be used as homework, for self-study practice, or in a lab, as live lecture examples. The problems presented are short problem statements (like an exam) and immediate feedback is given in the browser. The idea for CodingBat came from experience with teaching CS at Stanford combined with seeing how students used unit-tests in more advanced courses. CodingBat has several coding exercises for recursion.

Zyante* eBooks have textual content, animations, and a few automatically assessed short answer questions. They do not have proficiency exercises nor visualization for student's code. Proficiency exercises (also called visual algorithm simulations) aim to verify that students understand how a given algorithm works by requiring them to simulate its behavior [25]. Zyante does not have comprehensive visualizations and examples for recursion.

Miller and Ranum's interactive eTextbook for Python programming [69], and *CS Circles* by Pritchard and Vasiga [79] are Python courses for novice programmers. Miller and Ranum produced a complete book that includes embedded video clips, active code blocks that can be edited within the book's browser page by the learner, and a code visualizer that allows a student to step forward and backward through example code while observing the state of program variables. Like OpenDSA, they use reStructuredText (ReST) and Sphinx (sphinx.pocoo.org) as their authoring system. Miller and Ranum's book runs on the Google App Engine and includes assessment activities requiring students to write small single function programming exercises. Their grading system is rudimentary and only provides students with simple pass/fail feedback upon completing an exercise. In our opinion, the most important thing missing from this effort is a wider variety of automated assessment with immediate feedback.

CS Circles is an exercise-centric online eTextbook for learning Python. It uses the WordPress Content Management System as its authoring tool, and the CodeMirror plugin † to allow in-browser code editing and automated program exercise assessment. CS Circles has only few simple recursion programming exercises.

Both Miller and Ranum's book and *CS Circles* use Guo's online Python tutor [42], an embeddable web program visualization for Python. The online Python tutor takes a python source code as input and outputs an *execution trace* of the program. The trace is an ordered list containing the state of the program at each line of code. Each execution point contains the line number of the code that is about to be executed, a map of global variables to their current values, an ordered list of stack frames with each frame containing a map of local variables to their current values, the state of the heap, and the program output up to the execution point. The trace is encoded in JSON format and sent to the user's browser for visualization via HTTP GET request. The backend can work on any webserver with CGI support or on the Google App Engine.

Both Miller and Ranum's book and *CS Circles* have a section on recursion. The recursion

*https://zybooks.zyante.com/

†https://codemirror.net/

section of Miller and Ranum's book has code writing exercises, a few multiple choice questions, and a visualization for a fractal drawing example. The recursion section of *CS Circles* [79] has only a few coding examples and code writing and completion exercises. Both the examples and exercises can be visualized. The visualization traces the code line by line.

Mozilla Thimble is a programming environment that helps users to learn HTML through allowing them to write their code on the right hand side of the browser and show their code output on the other side.

These e-Textbooks are a good step for moving from a traditional textbook to an online system that has a way for automatically assess and visualize students code. However, none of those systems comes from the study of the best pedagogical way to teach hard topics like recursion. They do not explicitly address misconceptions that students usually have in those topics. And they all lack sufficient programming practice exercises with sufficient feedback.

2.2 Teaching Recursion

2.2.1 Introduction

Most previous research on teaching recursion has focused on abstract discussions of the recursion concept and its control flow [70, 19, 33, 78, 93, 97, 117], comparing recursion to other disciplines [22, 59, 85, 115], new ways to view the recursion concept [19, 117, 29, 118, 19], and the use of visualization, animation and games to help students to understand recursion [8, 30, 41, 47, 99, 116, 105, 37, 100, 112, 16, 119, 21, 101]. Gordon [37] and Stephenson [100] argue that visual displays, as in the drawing of fractals, will assist students in understanding that computation could happen as a function goes into and comes out of recursion as well as at the limiting case. Some researchers use programming environments to visualize recursion in the student's code. Stern and Naish [101] showed that classification is a useful guide to picking an appropriate strategy when animating recursive algorithms. The way they classify a recursive algorithm is based on how the algorithm is dealing with the data structure it works on (e.g. insertion, navigation, etc).

2.2.2 Successful approaches

While previous research includes in-class experiments [8, 33, 93, 116, 105], we found two papers that included controlled experiments to provide statistically significant evidence that the proposed teaching methods improve student learning of recursion [105, 110]. Chaffin et al. [8] did no controlled experiment, but compared the scores of the pre-test and post-test given to the students before and after using a game. Their analysis showed statistically significant evidence that the proposed teaching methods improve student knowledge of recursion by comparing pre-test grades to post-test grades, but don't compare their method against "structured lecture" and textbook.

Tessler et al. [105] investigate a new method for teaching recursion in which students play Cargo-Bot to situate learning before they are formally taught recursion. Cargo-Bot is a video game for the Apple iPad in which users teach a robot how to move crates to a specified goal configuration by writing recursive programs in a lightweight visual programming language. The game was not designed originally for educational use. With the developers' permission, Tessler et al. rewrote Cargo-bot in JavaScript to make it accessible to all students with Internet access. They also modified the game to include user identification and tracking, so as to track the game play of individual students.

The experiments done by Tessler et al. tracked two groups of students. 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. The pre- and post-tests also contained several survey questions using a Likert scale to measure student engagement.

The analysis of the results of the writing portions of the pre-, mid-, and post-tests, in which students create their own recursive solutions, showed that students in the control group see a drop in performance from the pre-test to the mid-test, (i.e., after receiving direct instruction). After then playing Cargo-Bot, their scores increased by approximately 19.48%. By contrast, students in the experimental group experienced the greatest increase in performance between the pre- to mid-tests, when they played Cargo-Bot. After the subsequent direct instruction, their test scores increase by just 4.48%. Using a two-sample Student's t-test, it was shown that the learning gains (i.e., the difference in test scores) from the pre- to mid-tests for the experimental group are greater than those of the control group, and that students experience greater learning gains in their abilities to write recursive functions after playing Cargo-Bot, rather than from direct instruction. Very different results were observed on the tracing portion of the pre-, mid-, and post-tests. Students in both the control and experimental groups experience a decline in tracing performance from the pre- to mid-tests, then an increase from the mid- to post-tests. These results show that the new method of teaching recursion produces no significant difference in improving students' abilities to trace the execution of recursive functions. The authors have not showed the total gain of the pre- to post test on the two groups. Thus, playing Cargo-Bot significantly improves students' ability to write recursive functions, but it does not improve students' abilities to trace the execution of recursive functions. This is unsurprising, as Cargo-Bot does not explicitly involve code tracing; players instead use recursion at the conceptual problem-solving level, rather than as the procedural process. The pre-test survey results showed that students are confident in their understanding of recursion. It would have been instructive to have included these same questions on the post-test. The post-test survey results showed that the vast majority of students enjoy playing Cargo-Bot, and most are either neutral or confident in their ability to play Cargo-Bot.

The authors mentioned that the success of Cargo-bot might be because it helps students write a particular type of recursive function that matches a syntactic template for the function, but that it doesn't help build the skills to trace the recursive behavior of arbitrary functions.

Tung et al. [110] present a new approach to teaching recursion, using visualcode. Visualcode is a visual notation that uses coloured expressions and graphical environments to describe the execution of Scheme programs. Visualcode can generate a dynamic visual representation of a recursive evaluation, and can also explicitly present the unfolding of recursive calls and passing back of control from callers. RainbowScheme is a program visualization system that is designed to produce visualcode representations from the step-by-step execution of Scheme programs. The visualization technique introduced is based on the operational semantics of the Scheme programming language. Results of this study support the claim that asking students to view visual execution steps produced by RainbowScheme and later requiring them to reproduce those steps manually for similar problems can assist students to understand the concept of recursion efficiently. The study was conducted while students were taking an introductory programming course. The students were divided into two groups. The experimental group received instruction using visualcode. The control group received instruction without using visualcode. Each group had 21 students. A t-test was conducted to examine the differences between the means of the post-test for the two groups. The results showed that students in the visual group significantly outperformed those who were in the control group in both evaluation questions and programming questions. The average scores of the evaluation questions for the visualcode group and the control group were 5.10 and 1.67, respectively. The average scores of the programming questions for the visualcode group and the control group were 5.67 and 3.57, respectively. t-tests indicate that the visualcode group significantly outperformed the control group on all individual evaluation questions. For programming questions, the visualcode group performed significantly better than the control group.

Chaffin et al. [8] presented a novel 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. The game is designed to facilitate transfer of learning to writing real programs, while also providing for interactive visualizations. The overall design of the game involves completing three programming puzzles, helped by Ele, a programmable avatar for visualizing data to collect, and Cera, an in-game mentor, who instructs students as they progress through the game, with dialog. In the game, students first take a brief pretest to determine their understanding of recursion. Then students complete a basic "hello world" program to get used to the compiler interface. Then students walk their character using depth first traversal to collect Thoughts from the leaves of a binary tree. For level 2, students must correctly code the traversal for the left side of a DFS. After their code is written, Ele walks through the tree using the student code while Cera explains what the code is doing. For level 3, students code both the right and left DFS for the binary tree and navigate Ele through the binary tree using the keyboard and mouse. This time, visualization is provided for the stack calls made by the recursive algorithm. Once the player finishes the game, they take the final survey to complete their journey. An evaluation was done for the prototype using 43 students who were enrolled or had completed a Data structures and Algorithms course. The students were first given a pre-test for recursion related computing concepts. Then, they were asked to play the game for 40 minutes. Finally, the students were given a post test and took a survey about their experience. The post-test is similar to the pre-test, with the numbers and variables names changed in each question.

A pre- to post-test comparison was conducted for students taking both tests (N=16). With a p value of approximately 0.008, the authors showed there is a significant improvement in the post-test scores compared to the pre-test scores.

Cargo-bot used a visual programming language to teach students recursion. Other than Cargo-bot, we note that none of the previous work on teaching recursion has used programming exercises and automated assessment techniques to help the students understand the hard topics through doing more practice and receiving immediate feedback.

2.2.3 Models for understanding and representing recursion

Flow of control

The flow of control for a recursive function has two parts. Active flow refers to the forward passing of control where a programmer has explicitly called the function. Passive flow refers to the backward flow where control is automatically passed back to the function at the point the active flow has completed by reaching the limiting case [30]. So there are three important concepts that need to be understood by the student in order to write a recursive function, namely: active flow, limiting case, and passive flow.

Mental models for understanding recursion

Mental model is a term used by cognitive psychologists to describe the cognitive representation of knowledge [48]. Much research has sought to discover the mental models used by students in understanding recursion. Other researchers focus on the conceptual models that can be used to represent and teach recursion.

[91] and [38] classified recursion mental models into two main categories: viable and non-viable, where viable models are those that allow correct prediction of program behaviour. In [91] and [38] the mental models were identified from students' traces of the execution of recursive programs. A trace is a student's representation of the flow of control and the calculation of the solution for a recursive program. The mental models that have been identified for recursion are the copies model [50], analogy model [115], looping model [50], active model [38], step model [38], return value model [38], magic or syntactic model [50],

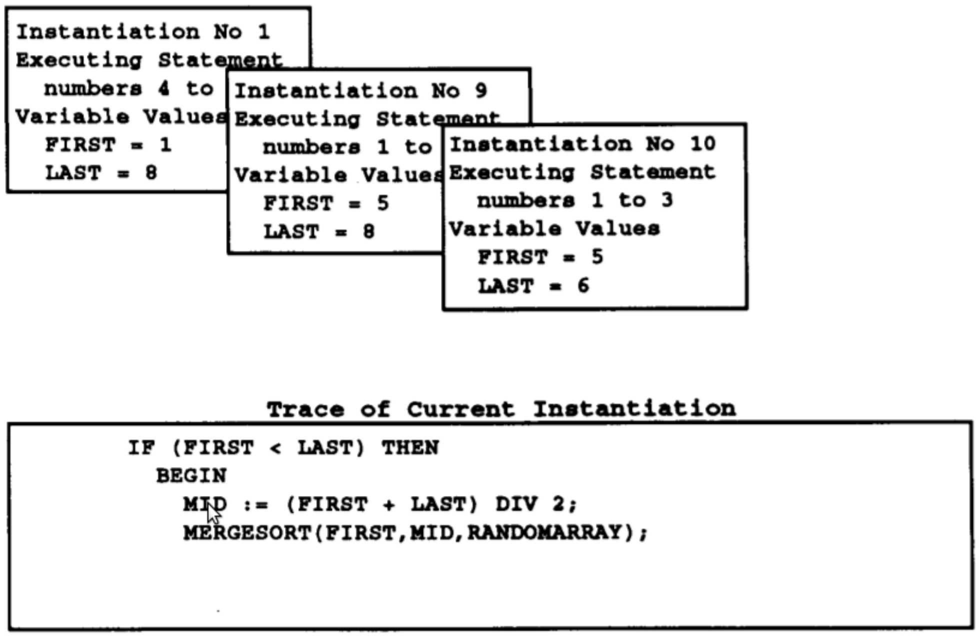
and algebraic model [38].

Kahney [50] presents a study on a sample of thirty university students who had done at most one introductory programming course. They showed that most of the students had developed one of the following incorrect mental models (27 out of the 30): looping, odd, or syntactic magic. This research has shown that the copies model is the only viable model and that it should be used for representing recursive functions. In addition, Sanders et al. [91] showed that the copies model is always viable. In other words, copies model is the only viable model that can be used to trace the behavior of a recursive function to understand it.

Conceptual Models for teaching recursion

[118] reported five conceptual models that have been widely used for teaching recursion. The first three can be categorized as concrete models and the remaining two as abstract models.

- Russian Dolls [13]: A Russian Doll can be taken apart into many successively smaller dolls of the same shape. It displays the process of invoking a smaller size of itself (recursive case) and eventually the recursive process stops when the last doll does not contain another (base case). This model is similar to the copies model in [116]. The copies model can be visualized as if you are seeing yourself in a mirror, in a mirror, in a mirror, etc. Each recursive call is a copy of the recursive function call with different parameters. One good characteristic of this model is that it can be visualized easily on a computer screen as overlaid windows and hence it is suitable for passive flow tracing. Figure 2.1 shows a copies model representation for a recursive mergesort algorithm for the data set 2, 5, 7, 6, 4, 3, 1, 8.
- Process Tracing [55]: This approach focuses on tracing the process generated by recursive functions, that is, how recursive functions work. This model is clearly a concrete model, but the degree of concreteness varies depending on the method used in tracing the process. One method that can be used for tracing is the tree representation [59]. It represents each recursive call as a node in a tree. This representation is suitable for novice programmers. However, the representation of a tree on a screen with a reasonable number of details on each node can only be feasible for small trees.
- Stack Simulation [40]: Recursion is introduced in terms of computer architectures for execution of recursive programs. Calls to functions or procedures are traced with explicit reference to the system stack mechanism that is used when implementing recursion in a programming language.
- Mathematical Induction ([111], [23]): This approach introduces recursion in terms of the mathematical basis for its correctness; that is, proof by induction.
- Structure Template [77]. This model provides novice programmers with samples of recursive programs and describes the base cases and recursive cases. The student solves the recursive problem by filling in the slots of base case(s) and recursive case(s) in a structural template. This model is similar to the graphical recursive structure



```
Instantiation No 1
Executing Statement
  numbers 4 to
Variable Values
  FIRST = 1
  LAST = 8
```

```
Instantiation No 9
Executing Statement
  numbers 1 to
Variable Values
  FIRST = 5
  LAST = 8
```

```
Instantiation No 10
Executing Statement
  numbers 1 to 3
Variable Values
  FIRST = 5
  LAST = 6
```

Trace of Current Instantiation

```
IF (FIRST < LAST) THEN
  BEGIN
    MID := (FIRST + LAST) DIV 2;
    MERGESORT(FIRST,MID,RANDOMARRAY);
```

Figure 2.1: Copies model representation for a recursive mergesort algorithm for the data set 2, 5, 7, 6, 4, 3, 1, 8

model in [34]. In this model, the recursion concept is applied to objects rather than to programs or to the process of their execution. Recognition of a "recursive structure" for an object can lead the beginning student to a "recursive description" of that object, and that description may lead him or her to a program that emulates, in one way or another, the recursion associated with the given object. The program can be viewed as emulating (generating, processing, computing, or simulating) a given object. The object can be the output of the program or even the process created by the program. According to this approach, when thinking about recursion, the beginning student can start from the programming task. When we consider recursive graphics such as fractals, we may start with another characterization. In this context, unlike the context of functions, the programming task is given by an instance of a sequence. Therefore, the beginning student must first observe the given drawing as an instance of a repeating pattern. She or he may do so by discovering a recursive structure geometric object: an object (e.g., a geometric object) has a recursive structure when it can be defined as a growing object, based on a fixed law of growth. This definition applies to many fractals and other types of recursive geometric patterns. This model requires that the student abstract the recursive nature of a graphical structure. The problem with this model is that it requires the ability to abstract, so it is unlikely to be used by novice programmers.

Few studies have been done in the field of programming using conceptual models. Mayer and Bayman ([65] , [66]) provide experimental evidence that concrete models promote learning.

However, Mayer and Bayman's studies did not explore the complex conceptual knowledge involved in large program segments such as the concept of a loop or the concept of a data structure. Nor did they compare the effects of different types of conceptual models.

2.2.4 Types of recursive algorithms

There are several types of recursive algorithms. Rubio categorizes them according to the number and the type of the recursive function calls within a procedure [84]. A common classification distinguishes the following types: linear, tail, binary, multiple (or exponential), nested, and mutual recursion [86].

- Linear recursion: This is the most commonly used type of recursion. Here, a function calls itself in a simple manner and terminates when reaching the limiting case. The factorial function is a good example of linear recursion. Another example of a linear recursive function would be one to compute the square root of a number using Newton's method. A third example is the recursive binary search. An example of a factorial recursive function is shown below:

  ```
  int factorial(int n){
    int result;
    if(n==1)
      return 1;
    result = factorial(n-1) * n;
    return result;
  }
  ```

- Tail recursion: This is a form of linear recursion. In tail recursion, the recursive call is the last thing that the function does. Often, the value of the recursive call is returned. As such, tail recursive functions can often be easily implemented in an iterative manner. By taking out the recursive call and replacing it with a loop, the same effect can generally be achieved. In fact, a good compiler can recognize tail recursion and convert it to iteration in order to optimize the performance of the code. A good example of a tail recursive function is a function to compute the Greatest Common Divisor of two numbers. An example of a greatest common divisor recursive function is shown below:

  ```
  int gcd (int x,int y){
    if ((x % y) == 0)
      return y;
    else
      return gcd (y, x % y);
  }
  ```

- Binary recursion: Some recursive functions don't just have one recursive call, they have two (or more). Binary Recursion is a process where a function is called at least twice. It is used most commonly in data structure operations for trees such as traversal, finding height, merging, etc. An example of a Fibonacci recursive function is shown below:

  ```
  long fibonacci(int n){
    if (n > 2)
      return fibonacci(n-1) + fibonacci(n-2);
    else
      return 1;
  }
  ```

- Exponential recursion: Where the number of functions calls is exponential in relation to the size of the data passed.Example of exponential recursion are a function to compute all of the permutations of a data set, and a function to recursively compute the Fibonacci numbers.
- Nested recursion: In this type of recursion, one of the arguments to the recursive function is the recursive function itself. These functions tend to grow extremely fast. An example of nested recursion is Ackerman's function.
- Mutual recursion: A recursive function doesn't necessarily need to call itself. Some recursive functions work in pairs or even larger groups. Function A calls function B, and function B calls function A. A good example is a pair of functions to determine whether an integer is even or odd as shown below:

  ```
  boolean even(int number){
    if( number == 0 )
      return true;
    else
      return odd(abs(number)-1)
  }

  boolean odd( int number ){
    if( number == 0 )
      return false;
    else
      return even(abs(number)-1);
  }
  ```

In [87] a methodology was proposed for deriving tail recursion functions that is based on declarative programming and the concept of function generalization. These approaches allow students to avoid iterative thinking. In [30] a visualization aid was introduced to help students understand linear recursion using the copies model and a study was made to identify various errors and misconceptions not related to linear recursion that can affect student answer to recursion question like misconceptions related to variable updating, memory storage and conditional statements evaluation.

2.2.5 Problems that students have with understanding recursion

Previous research has determined the most common problems that lead to students misunderstanding recursion. Some research suggests solutions and recommendations to solve those problems.

Most commonly, students struggle with the unfamiliarity of recursive activities [4], visualization of the program execution [40], backward flow of control after reaching the base case (i.e., passive control flow) [93], comparison to loop structures [4], and the lack of everyday analogies [77].

[70] discusses an investigation intended to address the learning of recursion in a multidimensional perspective, where the dimensions correspond to different types of competence relevant to programming. This research tried to identify specific learning obstacles to understanding recursion. It was concluded that neither the language syntax nor the computation model are crucial for the learning process and that we should spend more effort on the declarative aspects implied by programming, since "The key to comprehending any form of abstraction, including recursion, is to focus on the what and down play the how"[98].

[50] and [53] concluded that novices easily understand tail recursion using procedures but have difficulty with understanding tail recursion with functions and embedded recursion. This is because students have difficulty understanding the control mechanism of recursion (especially passive flow of control) and do not have proper mental models for recursive processes.

[117] highlighted the fact that understanding the recursion process is not related to the trace mental model. Previous research has indicated that supposedly simple tail recursive algorithms should be avoided because these can make students think of recursion as a "loop"[91] because the recursive call is the last step in the function and hence the function is repeating itself again.

In [90] it was claimed that a key factor in mastering recursion is understanding how the program moves from active control to the base case and then to the passive control in recursive functions. The complexity of the flow of control mechanism makes it a difficult concept for students to comprehend. The results show that in most cases students have some difficulty with the active flow, are confused about the passive flow, and have misconceptions about the limiting case [30]. It was recommended that in order to teach students to understand passive flow better, more explicitly embedded recursive algorithms where there are executable lines of code both before and after the recursive call should be used as the first examples in teaching the concept. In addition, different types of recursive algorithms and different approaches to teaching recursion should also be investigated. The authors recommended that we should give the students tasks to trace a program where the active flow simply reduces the problem to the limiting case, the limiting case is essentially a switch from the active to the passive flow without returning any values, and the main work of the program is done in the passive flow. If a student was able to generate the correct output then they would be showing understanding of the flow of control. They presented a task given to their students for this purpose. They asked the students to determine the output of a given recursive Python program which uses Python's turtle module. The program takes a number and a distance value as input. Each student was asked to evaluate the given program for n equals 10 and distance equals 100. They were given a pencil and paper that they could use to follow the flow of control of the program and draw the output. The expected pattern output drawn by the given program is a square spiral.

Since grading students answers for such a task is time consuming this implies that it would be beneficial if we have an automated way to detect what confuses the student from the student's answers.

[93] suggested that we should involve tracing methods or explicitly stating the function calls of a recursive algorithm to aid students when teaching recursion. They provided students with a mechanical means of following the execution of a recursive algorithm.

Lewis [60] studied the variation in students' successful attempts to trace linear recursion. Lewis has shown four modes of tracing linear recursion that may require or facilitate a particular understanding of recursion. The author then suggested to use his findings in building a representations for tracking execution to teach students how to accurately trace linear recursion execution.

Murphy et al. [71] conducted a goal-plan analysis to find out the plans used by students when writing a recursive method to count the number of nodes that has exactly one child in a Binary Search Tree. Students were required to write a method that traverses the tree and counts the nodes. Analysis of the students' answers showed that more than half of the students tested for the base case before it was actually reached in order to avoid making recursive calls. Even the students who did not do that had difficulty with bases cases, misplaced calculations, and missed recursive calls. Murphy et al. found that their findings are useful for designing questions or homeworks, and that instructors should address this in class. This work does not address specific misconceptions in binary trees, but rather is related to general problems with recursion.

Vilner et al. [113] recommended that when teaching recursion we should use as many examples as possible, starting with visual examples. The authors emphasized the importance of providing as many different problems as possible to convince students about the importance of recursion. Also, it is important to continue showing examples of recursive algorithms after the concept has been introduced in the chapter on recursion. Recursion can be shown when teaching linked lists, arrays, searching and sorting algorithms, and binary trees.

Recent studies suggest to differentiate between students based on their skills through an adaptive teaching strategy that classifies students [81], and to see the effect of introducing recursion in the context of recursively-defined objects, such as lists and binary trees rather than in the context of mathematical functions [67].

2.3 Automated Assessment

Previous research has shown that automated assessment of programming exercises is beneficial for both students and instructors [89, 68]. [89] has implemented a system called Scheme-robo for assessing programming exercises written in the functional programming language Scheme. The system assesses individual procedures instead of complete programs, and provides feedback to the students. The system has been in production use in an introductory programming course with some 350 students for two years. The system was helpful as it helps 350 students to practice at least 5 exercises on average per week without putting work on the instructor to correct their answers. In addition, students get immediate feedback on their answers which makes them learn from their mistakes. [89] conducted a survey showing that 80 percent of the students thought that automatic assessment in general is a good or an excellent idea. [68] studies the feasibility of automatically assessed exercises. The authors recommend using both in-class and automatically assessed exercises instead of using only one of these. We should take into consideration that having both in-class and automatically assessed exercises would be at the expense of either consuming more time from the student or sacrificing some in-class activities or content.

This section presents approaches used for automated assessment of programming exercises. There are three types of automated assessment for programming exercises: output-based (also known as dynamic), static, and trace-based assessment. Output-based program assessment runs the program against test cases, then compares the output of the student's code against the output of the model answer. Static assessment assesses of the student's code without running it. The goal is to find if the code fulfils some quality metrics (e.g. variable naming, comments, good programming practice, etc.). Trace-based assessment runs the program to make sure that certain variable values/states are changing according to the requirements. There are multiple ways of doing trace-based assessment based on the assessment requirements. Trace-based assessment is more challenging because many variations may exist in a student program that satisfies the requirements specified in the problem statement. Previous work on each type of assessment is discussed next.

Output-Based Program Assessment

There has been much work on output-based program assessment. For example, [44, 51, 62, 89, 6, 61] use unit testing to evaluate student code. In [44] the assessment is done on the server side. This work provides an assessment of only whether or not the program, or parts of the program, produces the correct answer. Static analysis is (optionally) done to the student code first to detect possible bugs, dead code, and suboptimal or overly complicated code. The instructor should still examine the students' code to provide advice and assessment of design and implementation style. In [51] assessment is done on the client-side. The advantage of client-side assessment is that the installation and sand-boxing of a server are not required. On the other hand, one concern with client-side assessment is that the exercises can only be used for self study because the grades sent from the browser can be tampered with. Another problem is that the necessary programming environment must be available on the client.

Static Assessment

This type of assessment measures code quality, finds bugs, and ensures that the student's answer is following good programming techniques. Truong et al. [109] target fill-in-the blank problems, where a student is given a piece of code that has missing commands and is then asked to complete the code. The aim of their work is to assess code quality, not correctness. Software metrics were used to assess the quality of the student's code. A drawback of the proposed approach is that it does not take the different syntactic forms of a model solution into account. Moreover, the similarity check considers only the outline of a solution and not its details.

Trace-Based Program Assessment

There are fewer efforts dedicated to the trace-based program assessment [120, 31, 108, 103] due to its complexity.

In [120] a transformation-based approach is implemented to automate the diagnosis of student programs for programming tutoring systems. The main techniques presented are program standardization and semantic-level program matching. This is done to compare the student's answer to a model program.

[31] assesses student code by matching it to a model answer to ensure that the student's code is following good programming techniques. This system categorizes student's code into one of four predefined categories (good, good with modifications, imperfect, and incorrect), which reflects whether it follows good programming techniques. This system does not provide feedback to students to show the problems in their code. The Pass system [108] checks if the student code matches the model answer. A drawback of the system that it overly constraints student answers. For example, the system considers the use of any helper function to be incorrect.

Trying to understand what a program is doing is called program comprehension. This is related to trace-based automated assessment. Program comprehension is defined as the process of acquiring knowledge about a computer program. It is often used to acquire more knowledge about a program in order to enable activities like bug correction, code enhancement, reuse, and documentation ([88] , [43] , [18]). Program comprehension is sometimes called reverse engineering [73]. Automatic program comprehension has been studied from two different points of views: understanding the functionality of the program and understanding the program structure. Most of the studies fall in the first category.

2.4 Concept Inventories

2.4.1 Introduction

A Concept Inventory (CI) is a test that can classify an examinee to whether they think in accordance with accepted conceptions on a body of knowledge or in accordance with common misconceptions [83].

To be considered a successful and valid instrument, a CI must be approved by content experts. A CI is not a comprehensive test of everything a student should know about a topic after instruction [45]. Rather, CIs selectively test only critical concepts of a topic [83], since these are required to be considered to have mastered the topic.

CIs have been successfully developed and used in STEM disciplines like Physics [92], Chemistry [57] and Biology [17] to drive discipline-specific education research and pedagogical reforms [104, 2]. For example, in Physics, the Force Concept Inventory (FCI) showed gaps between how students and instructors think about concepts related to mechanics [92].

In Computer Science, the development of concept inventories is growing. The next subsection presents efforts in Computer Science concept inventory development.

2.4.2 Computer Science CIs

Efforts to develop CIs have been done for discrete math [2], digital logic [46], operating systems [3, 114], introductory programming courses [49], algorithms and data structures [15, 76], binary search trees [52], and object oriented programming [80].

Kaczmarczyk et al. [49] worked on finding student misconceptions in core CS1-level programming course concepts. Based on a Delphi process [14], the authors gathered from experts 30 concepts that experts think are the most difficult ones covered in CS1. The authors selected ten concepts as their initial focus of interest. The selected concepts are: memory model, references and pointers, primitive and reference type variables, control flow, iteration and loops, types, conditionals, assignment statements, arrays, and operator precedence. The authors designed a test of 18 questions covering the concepts of interest. In order to make sure that the results are not problem dependent, the concepts were covered in at least two different variations. The authors believed that conducting student interviews would help them understand students misconceptions of the targeted concepts. Eleven undergraduate students participated in the interviews. The students at the time of the interviews were either currently or recently enrolled in a Computer Science introductory course. Each interview lasted about an hour and was audio and video recorded. In the interview, each student was asked to solve questions for all ten concepts. The purpose of the interviews were to reveal the misconceptions of the students and validate the expert's conclusions about the difficult concepts. The authors analyzed the student interviews and described in detail the misconceptions found in memory model representation and default value assignment of primitive values. As their future work the authors plan to obtain additional interviews and tests from multiple institutions. Then, a test inventory should be built and pilot tests should take place at multiple institutions. Finally, the test inventory should be enhanced based on the test results.

Danielsiek et al. [15] described the first results towards building a concept inventory for Algorithms and Data Structures. Their results are based on expert interviews and the analysis of 400 exams to identify the core concepts that are considered to be error prone. They conducted a pilot study to verify the misconceptions known from the literature and identify previously unknown misconceptions. They then extended their efforts to build an initial instrument to detect misconceptions related to algorithms and data structures [76]. In addition, they presented the results from a second study that aimed at assessing first-year student misconceptions. Their second study confirmed findings from the previous small-scale studies, but additionally broadened the scope of the topics.

Karpierz et al. [52] attempted to find misconceptions and design a concept inventory for Binary Search Trees and Hash Tables. The misconceptions identified were not related to recursion. Even the questions presented in the interviews had iterative code. The authors found student misconceptions by interviewing 9 instructors, showing them sample exam responses with the goal to understand how an expert reorganizes something important that the audience does not. In addition, the authors reviewed more than 200 exam problems. They analyzed exam and project code to find the most difficult problems. In addition, they interviewed 25 students who each solved two questions while thinking aloud. The authors found three topics for misconceptions: the possibility of duplicates in BSTs, conflation of Heaps and BSTs, and Hash table resizing. The authors have designed three multiple choice questions to recognize those misconceptions. As a future work, the authors plan to validate the concept inventory by giving it to students at different institutions.

Ragonis and Ben Ari [80] presented an initial effort to identify misconceptions and difficulties in object oriented programming (OOP). The authors gathered data during two academic years from students studying OOP in tenth grade CS. The data gathered included home works, lab exercises, tests and projects. They used this data to identify a comprehensive categorized list of misconceptions and difficulties in OOP understanding.

Taylor [104] presented the most recent survey paper on Computer Science CIs. Taylor recommended building CIs for topics that should evaluate student's ability to engage in processes such as code analysis, program design, program modification, and testing. He mentioned that these aspects of learning are difficult to assess, and some aspects of understanding are hard to evaluate. The same was also concluded by Zingaro [122] who stated that it is hard to evaluate some aspects of understanding with an example being the difficulty of grading traditional code writing exercises.

2.4.3 Building a CI

This section presents the traditional steps that should be followed to build a CI and the ways used for measuring a CI's reliability and validity [45, 72, 35, 46, 58].

1. **Choose concepts** (set the scope): First a set of concepts is chosen by the CI developers to define the CI's scope. To assure that the CI is a valid assessment tool, many domain experts must acknowledge that the tool assesses the right content, and that it does in fact assess what it claims to assess. By involving expert opinion from the beginning of the CI development process, we can trust that the designed CI assesses core concepts and that it has appropriate content validity [1].
2. **Identify misconceptions**: Instructors and students can be interviewed to identify the specific sub-topics that students struggle to understand. Instructors can identify students' misconceptions from their teaching and exam-marking experience. Students can also be helpful in identifying their confusion about a certain topic [1].
3. **Write CI items and draft the CI** (write the questions): The CI developers should use the misconceptions identified from the previous step to formulate the CI questions. The questions could be multiple choice (MCQ), or any other type of question where incorrect answers can be used to identify the associated misconception. For the sake of reliability, the CI would ideally test every concept multiple times [7]. After writing questions for the initial CI, refinement and validation are done through two feedback cycles: the student feedback cycle and the expert feedback cycle.
4. **Student feedback cycle**: CI developers should give the CI to students and analyze the quality of the CI through interviews and statistical analysis. The interviews should ask students about the clarity of the questions and the answer choices (for MCQs) and find out if the students are truly solving the questions wrongly when they have the targeted misconception. In this step the reliability of the CI is to be measured to assess the prevalence of various misconceptions, and explore the data for differences in performance between sample populations. The CI should be revised and improved based on these analyses before repeating this cycle.
5. **Expert feedback cycle**: The CI content and individual items are evaluated by experts. The opinions from a diverse group of experts can reach consensus by using a Delphi process [14], an approach that has been used to develop previous CIs [35, 39, 102].
6. **Iterate**: The above sequence of steps could be repeated many times until a reliable

and valid CI is achieved. After each iteration, the CI is revised and modified to do a better job of evaluating student misconceptions, and the reliability and validity are measured.

Measuring CI's reliability and validity

Reliability

Reliability of a CI is usually estimated by three methods: test-retest reliability, split-half reliability, and the Cronbach alpha.

In the test-retest method, the reliability of the CI is measured by giving students the CI multiple times in close succession [1]. Test-retest is not usually done because it is a time consuming process and the students can learn little by taking the instrument multiple times, so its results may not be accurate.

Split-half reliability splits the test into two halves and treats each sub-test as a separate instance of the instrument. An estimate of the total reliability is made by building a correlation between the observed scores on the two sub-tests.

The most commonly used method is Cronbach alpha, which finds the average split-half reliability of every possible set of sub-tests. The Cronbach alpha value ranges from -1 to 1 like a correlation coefficient. A cut-off value is selected for alpha above which the CI is considered to be reliable. For example, [45] mentions a cut-off of 0.70 for the alpha value, because a high level of reliability was required. CIs need a high level of reliability to be used as a research instrument. However, some inconsistency can be acceptable since students are inconsistent when they apply their conceptual knowledge.

Validity

The validity of an instrument can be estimated by correlating the observed scores of a newly created instrument with the observed scores of an accepted instrument [1]. If there is no currently accepted instrument to measure the true score of a topic, statistical methods cannot be used to estimate the validity. Statistical estimates for the reliability for the instrument can potentially invalidate an instrument. As the reliability of an instrument decreases, the validity of the instrument also decreases. If the CI has a Cronbach alpha value below the selected cut-off, then it should not be considered as valid.

Validity can also be established in some cases through face validity and content validity [1]. Face validity exists if the typical person who is familiar with the material believes that the instrument measures the true score at first glance. Face validity must be done along with content validity to ensure the instrument's validity. Content validity is done by systematically polling the opinions of experts to see if they believe that the instrument measures the true score [1]. To test the validity of an instrument, its developers must clearly define what the instrument measures.

Chapter 3

Requirements Gathering

This chapter presents the requirements gathering process for building the recursion tutorials and concept inventory. First, we present our findings from surveys 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 have not used RecurTutor). We study confidence because research shows [74, 20, 94] 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 show the different skills required to write and trace a recursive function as determined in previous research [9, 5]. We used these skills as one of the basic principles in building our basic recursion tutorial. Last, we show the common misconceptions in basic recursion that we have found through analyzing 8000 recursion questions responses.

3.1 Instructor Surveys

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

Our analysis of the survey results answers the following research question:

Does the amount of time that students spend in typical classes learning and practicing recursion match the amount of time that instructors believe to be required to learn and understand this topic?

Participants: Participants were instructors who have at least one year of recursion teaching experience.

Materials and procedure: We received survey responses from 14 respondants (of 25 contacted) with at least one year of recursion teaching experience, regarding their views on the recursion course. The instructors belong to 6 different institutions in 3 different countries.

The instructor survey questions were as follows:

1. Briefly describe the course are you 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 to 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 home works 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?

Results: The results are shown in Table 3.1. The key findings from the surveys are that instructors spent on average 7 hours per semester in class covering recursion, and expected that students spend 10 hours on recursion outside of class.

Table 3.1: Instructors survey responses on time on recursion

Question	**Count**	**Average**
Course Level	CS2: 11 CS3 : 3	N/A
Background Required	None: 12 Basic: 2	N/A
Time on Recursion in Class	5 to 10 hrs: 12 Unknown: 2	7 hrs
Time required out-of-class	4 to 7 hrs: 2 8 to 27: 12	10 hrs
Students need more time out-of-class	Yes: 14 No: 0	N/A

3.2 Student Surveys

Our goal from conducting students surveys is to determine the time that students actually spend on recursion and their confidence level when using typical instruction

Participants: The participants were students enrolled in CS2114 Software Design and Data Structures in Spring 2014 at Virginia Tech. The students had not used our recursion tutorial, but have been assigned Coding Bat [75] recursion programming exercises.

Materials and procedure: During the last lab session of CS 2114, students were given a paper survey regarding their experience with learning recursion. A total of 54 students filled in the survey and returned it back 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 spent 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)

Results: The results are shown in Table 3.2. The key findings from the surveys indicate that the students spent a 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 an average of 10 hours outside of class.

Table 3.2: Spring 2014 students survey responses on time for recursion

Question	**Mean**
Time on Recursion out-of-class	4 hrs
Time on Coding Bat	1.9 hrs
Confidence level	2.5

From the surveys results, we confirm that students do not spend enough time out-of-class practicing recursion. The instructors unanimously felt that students were not spending enough time.

3.3 Skills required to read and write recursive code

Almost all of the previous evaluations done on teaching recursion asked the students to solve both code writing and code-tracing problems [8, 33, 93, 116, 105] in the pre and post tests. However, prior research on teaching recursion has not addressed the differences between the skills needed for code writing versus code tracing. In this section, we address the difference between the skills required to write a recursive function versus those required to read and understand a recursive function.

We agree with Michelene et al. [9] that a successful approach to writing a recursive function comes from thinking in a top-down manner. This successful approach is based on not worrying about how the recursive call solves the sub-problem. We teach students to simply accept that it will solve it correctly, and use this result to in turn correctly solve the original problem. For example, if the student is asked to compute $n!$ recursively, he should think in the following way:

- Know the mathematical function for computing the factorial: $n! = n * (n-1)!$
- To compute $n!$ it is required to compute $(n-1)!$. So multiply n by whatever returned from the recursive call of computing $(n-1)!$.
- Know the simplest case: $0! = 1$. The recursive calls will stop when n reaches to 0.

On the other hand, when it is required to read or trace a recursive function, we agree with Bhuiyan et al. [5] that the most useful and traditional approach to think about it is the stack model. 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.

3.4 Driving Hypothesis

We hypothesize that difficult programming concepts like recursion are best learned by an approach that provides a lot of practice exercises, and that students will achieve a better understanding of recursion through this approach.

From our initial surveys we have found that the traditional instructional process, as reported by the instructors, was failing in getting students to spend enough time on recursion. So one goal for our tutorial is to force longer (proactive) engagement to get students up to the required time levels reported by the instructors. The traditional instruction has a defect in the level of instructional engagement. So the recursion tutorial can make students spend more productive time engaging through doing practice and interacting with visualizations.

The instructor and student survey results set requirements for the basic recursion tutorial estimates of the time that students ought to spend on recursion out of class (10 hours on the average). To get students to spend this time in a proactive way, we thought about engaging them through practice exercises and visualizations that address their misconceptions. The difference in the skills required to write and trace a recursive function also sets a requirement on how the recursion tutorial should be organized and ordered.

3.5 Identify Basic Recursion Misconceptions

One of the important requirements for building both the recursion tutorial and the recursion concept inventory is to first find student common misconceptions.

To find out student misconceptions, typically, instructors and student interviews are conducted. We have sent the invitation email shown in Appendix G to 10 students attending CS2114 Data Structure and Software Design during Spring 2014 at Virginia Tech asking them to come for interviews. We received a positive reply from two. We interviewed these two students. Participation was voluntary and records were stripped of identification after the interviews were completed. The students signed a consent form before starting the interview. The interview was audio recorded and the students were made aware of that. The students solved 8 recursion tracing and one code writing exercises. Section I.1 in Appendix I shows the interview questions and the interview transcripts. Since student participation was low, it did not help us in finding student misconceptions. The common misconception found in the answers of both students interviewed was related to backward flow where the students did not understand what happens to information after the recursive call.

In addition to the student interviews, we also used test answers and literature to find student misconceptions, we analyzed approximately 8000 responses for recursion questions given to students over three semesters in pre-test, post-test, mid-term, or final exams of CS2114 Data Structures and Software Design to find out more common misconceptions. Table 3.3 shows the number of students and recursion questions on each test.

Table 3.3: Number of students and number of recursion questions per each exams.

Exam	**Number of Students**	**Number of 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

We have chosen to present our findings from the interviews and the analysis of student answers as a list of misconceptions and difficulties, inspired by Ragonis and Ben Ari's work on object oriented programming [80]. 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.

The following shows the common misconceptions and difficulties that we found, categorized by the topic related. We give each an identification tag for use in our analysis below.

Backward Flow

1. Misconception: No statements after the recursive call will execute. [BFneverExecute]
2. Misconception: Statements that come after the recursive call will execute before the recursive call is executed. [BFexecuteBefore]

Infinite recursion

3. 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. [InfiniteExecution]

Recursive call

4. Difficulty: Cannot formulate a recursive call that eventually reaches the base case. [RCwrite]
5. Misconception: A value will be returned from a recursive call even if the `return` keyword is omitted. [RCnoReturnRequired]
6. Misconception: All recursive calls require the `return` keyword even if the recursive function does not return a value. [RCreturnIsRequired]

Base case

7. 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. So the students has difficulty recognizing whether the recursive call or the base case is executed when tracing code. [BCbeforeRecursiveCase]
8. Misconception: The base case action must always return a constant, not a variable. [BCactionReturnConstant]
9. Misconception: The base case condition must always check a variable against a constant, not against another variable. [BCcheckAganistConstant]
10. 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. [BCwrite]
11. 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. [BCevaluation]

Updating variables

12. 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. [GlobalVariable]

The recursion misconceptions identified help us to frame the exercises, visualizations and prose to target those misconceptions.

Chapter 4

RecurTutor

In this chapter we discuss the basic recursion tutorial (RecurTutor) content and infrastructure.

4.1 Introduction

We have created the RecurTutor tutorial to teach basic recursion in CS2 courses. RecurTutor uses a new pedagogical approach based on greater student practice and interaction with the material than has previously been possible. The tutorial provides automatic assessment for the practice exercises, giving immediate feedback without putting additional grading burden on the instructor. The tutorial exposes students to a large number of selected interactive exercises, allowing them to practice a large number and variety of small-scale programming exercises. Both the examples and the exercises address common misconceptions and the various skills required to learn recursion. Students practice those exercises within the framework of tutorial text and visualizations for basic recursion.

No previous work on teaching recursion has adopted the pedagogical model of combining the material and substantial practice with automated assessment and feedback to the students. Our tutorial combines textual content and visualizations with exercises that are designed to avoid or expose common misconceptions.

We want students to achieve a better understanding of recursion, which we claim will come with the increased interaction with the large number of examples and exercises provided. We take "Practice makes it perfect"as our operating assumption, which means that the best way to learn is to work many practice problems that have been selected to gradually move students through the various subskills that are required for mastery [10].

4.2 The Tutorial Content

The tutorial content was reviewed and fine tuned by five instructors. Each of the instructors has more than ten years experience with teaching recursion. A full listing of the basic recursion misconceptions addressed in the tutorial can be found in Section 3.5 .

The tutorial is divided into the following modules:

1. Introduction: Focuses on the concept of abstraction in recursion.
2. Writing a recursive function: Shows the basic steps for writing a recursive function. It also shows different correct equivalent versions of a recursive function and the differences between them.
3. Code Completion Practice Exercises: Each exercise shows an incomplete recursive method and asks the student to complete the function by adding the missing base case, base case action, recursive case, or recursive call, so that the given function fulfills a certain requirement.
4. Writing a more sophisticated recursive function: Advances the student to the idea of having multiple base cases and recursive calls in a recursive function.
5. Harder Code Completion Practice Exercises: Each exercise shows an incomplete recursive method and asks the student to complete the function by adding the missing base cases, base cases actions, recursive cases, or recursive calls so that the given method fulfills a certain requirement. The functions shown in the exercises have more than base case and recursive calls to complete.
6. Writing Practice Exercises: Each exercise gives the student a problem to solve recursively. The student is asked to write a single function, typically five to ten lines of code, that solves the given problem.
7. Tracing recursive code: Shows the students, through visualizations, how to trace a recursive function.
8. Tracing Practice Exercises: Each exercise asks the student to trace a given function and find out the return value or the error in the recursive call or the base case in the function. Those exercises are fill in the blanks and multiple choice exercises.
9. Summary Exercises: Asks non-programming questions about the concept of recursion. Those are multiple choice exercises.

Given the writing and tracing skills presented in Section 3.3, our tutorial is intended to train students on both required skills to master recursion. We train the student on the writing skills before the tracing skills because we believe learning to write a recursive function first will help the students learn abstraction, which is a key factor in understanding recursion. In RecurTutor, we ask the student to solve 19 writing exercises, 17 tracing exercises and 9 non-programming questions.

RecurTutor is a chapter in the Data Structures and Software Design CS2114 online book in OpenDSA. Figure 4.10 shows RecurTutor as Chapter 6 in the CS2114 online book. An example lesson in RecurTutor is shown in Figure 4.11.

4.2.1 Textual content

We collect and mix the best features of the well-known and interesting recursion teaching books and online content from [82], [27] and [26].

We tried to minimize the textual content as much as possible, as we know from our previous experience [25] that students tend to skip the prose or give it less attention and jump to the exercises.

4.2.2 Visualizations

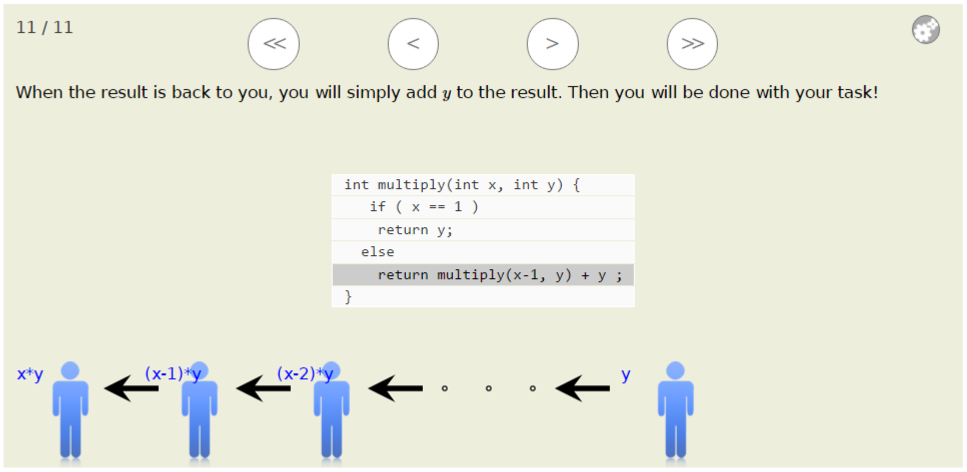
We have implemented three types of visualizations, introductory, writing and tracing visualizations. Introductory visualizations focuses on the abstraction of the concept of recursion. The writing visualizations teaches the student using examples how to write a recursive function. The tracing visualizations teaches the student using examples how to trace recursive functions. The visualizations focus on the basic recursion misconceptions and difficulties.

Introductory visualizations (example shown in Figure 4.1):

1. The first visualization focuses on the abstraction of recursion. This visualization uses the delegation process discussed by Edgington [19]. We agree with Edgington that presenting recursion as a particular form of task delegation is a good way to show the concept of abstraction in recursion. The visualization shows how to handle the task of multiplying two numbers x and y through delegation. The visualization shows how to do the multiplication task through delegating it to another friend, but makes it little bit easier by asking the friend to multiply $x - 1$ and y. So when the friend gives back that answer, then it is simple to add y to the result. The visualization emphasizes avoiding thinking about how the friend is going to do the task and to just focus on how the person who is delegating the task will do his own part.
2. The second visualization discusses the same multiplication problem, but this time looking deeper into the details of what the friend does when the task is delegated to him. It was made clear to the student that going to the deep details will be shown once but when writing their own recursive functions, they shouldn't worry about all of these details

Visualizations about writing a recursive function (example shown in Figure 4.2):

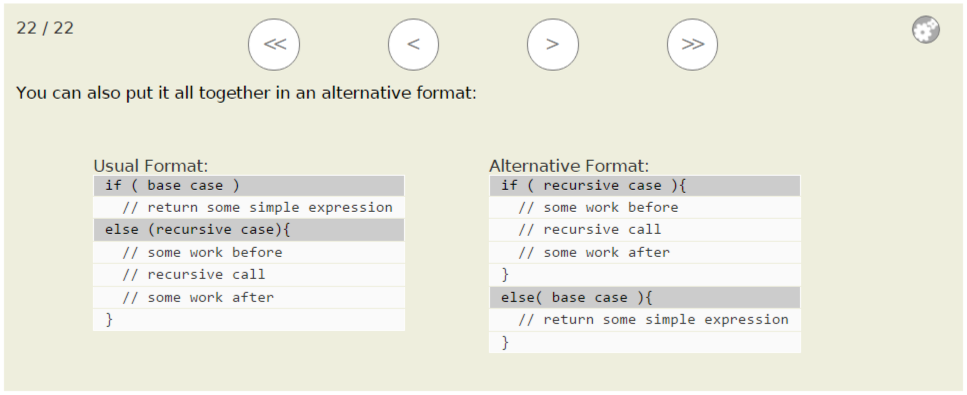
1. The first visualization shows the basic steps required to write any recursive function. The steps encourage the student to think in a top-down manner when writing a recursive function. The first step is to define and write the prototype of the function. The second, to write a sample function call. The third, to write the base case and the



When the result is back to you, you will simply add *y* to the result. Then you will be done with your task!

```
int multiply(int x, int y) {
   if ( x == 1 )
    return y;
  else
    return multiply(x-1, y) + y ;
}
```

Figure 4.1: Example of an introductory visualization



You can also put it all together in an alternative format:

Usual Format:

```
if ( base case )
  // return some simple expression
else (recursive case){
  // some work before
  // recursive call
  // some work after
}
```

Alternative Format:

```
if ( recursive case ){
  // some work before
  // recursive call
  // some work after
}
else( base case ){
  // return some simple expression
}
```

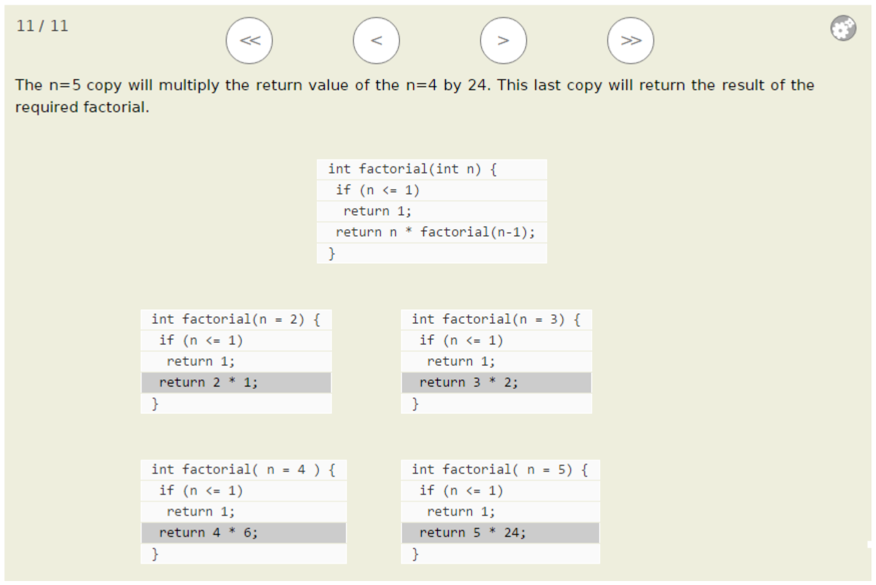
Figure 4.2: Example of a writing visualization

fourth, to think about the smaller versions of the problem. The example presented in the visualization recursively sums the values in an array.

2. The second visualization shows four alternative versions of implementing a recursive function of the sum example presented in the first visualization. The aim of the visualization is to show to the students that there are ways to write recursive functions that are syntactically different but functionally equivalent, so long as there is a recursive call that finds its way to the base case to end the recursive function when it is done.

Visualizations about tracing a recursive function (example shown in Figure 4.3):

1. The first visualization shows the winding and unwinding phases that the student has to consider when tracing a recursive function. The visualization emphasizes not forgetting the unwinding phase when tracing a recursive function as this known to be one of the common misconceptions. It also clarifies that the winding and unwinding is not really



The n=5 copy will multiply the return value of the n=4 by 24. This last copy will return the result of the required factorial.

```
int factorial(int n) {
 if (n <= 1)
  return 1;
 return n * factorial(n-1);
}
```

```
int factorial(n = 2) {
 if (n <= 1)
  return 1;
 return 2 * 1;
}
```

```
int factorial(n = 3) {
 if (n <= 1)
  return 1;
 return 3 * 2;
}
```

```
int factorial( n = 4 ) {
 if (n <= 1)
  return 1;
 return 4 * 6;
}
```

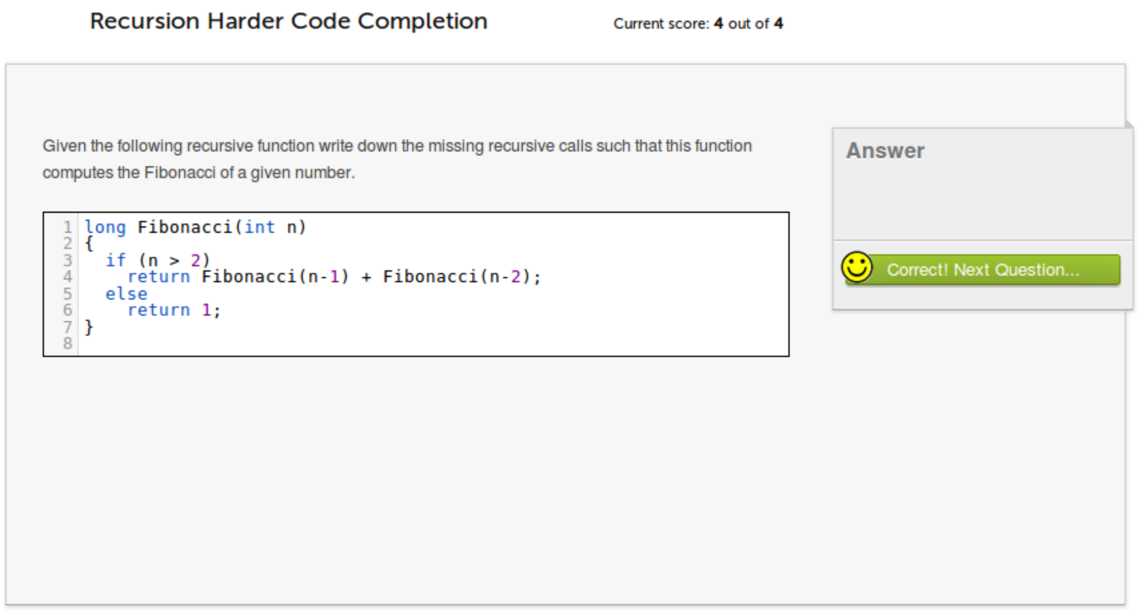
```
int factorial( n = 5) {
 if (n <= 1)
  return 1;
 return 5 * 24;
}
```

Figure 4.3: Example of a tracing visualization

special to recursion, as it occurs with any function.

2. The second visualization shows a tracing example for a simple recursive sum function.
3. The third visualization shows a tracing example of a factorial function. It follow the copies model as this was proven in [50] and [91] to be the most viable conceptual model.
4. The fourth visualization emphasis on the unwinding phase of a recursive function. It uses the sum example shown in the second visualization to clarify the unwinding phase and emphasizes not forgetting this phase while tracing a recursive function.
5. The fifth, sixth and seventh visualizations are called the domino effect visualizations. Those visualizations were suggested by [121] who recommended the use of the domino effect to model recursive computation based on computational semantics rather than on mathematical formalisms. The first visualization in the domino effect group shows an example of how to model the domino effect recursively. The second one illustrates the Domino effect as a solving technique to print positive integers from 1 to N recursively. The third visualization shows an example of the Domino effect as a solving technique to count the number of digits within an integer n recursively.
6. The last visualization shows a full trace of the Towers of Hanoi problems as an example of a problem that requires multiple recursive calls [64]. It traces the Towers of Hanoi code line by line while showing visually how the disks are moving from one support to the other and the call stack.

Each of the above mentioned visualizations is designed to address a basic recursion misconception or a set of misconceptions. Table A.3 in Appendix A shows a detailed description of the misconceptions covered by each visualization.


Recursion Harder Code Completion
Current score: 4 out of 4
Answer
Correct! Next Question...

Given the following recursive function write down the missing recursive calls such that this function computes the Fibonacci of a given number.

```
long Fibonacci(int n)
{
  if (n > 2)
    return Fibonacci(n-1) + Fibonacci(n-2);
  else
    return 1;
}
```

Figure 4.4: Code completion programming exercise with feedback on the correct answer.

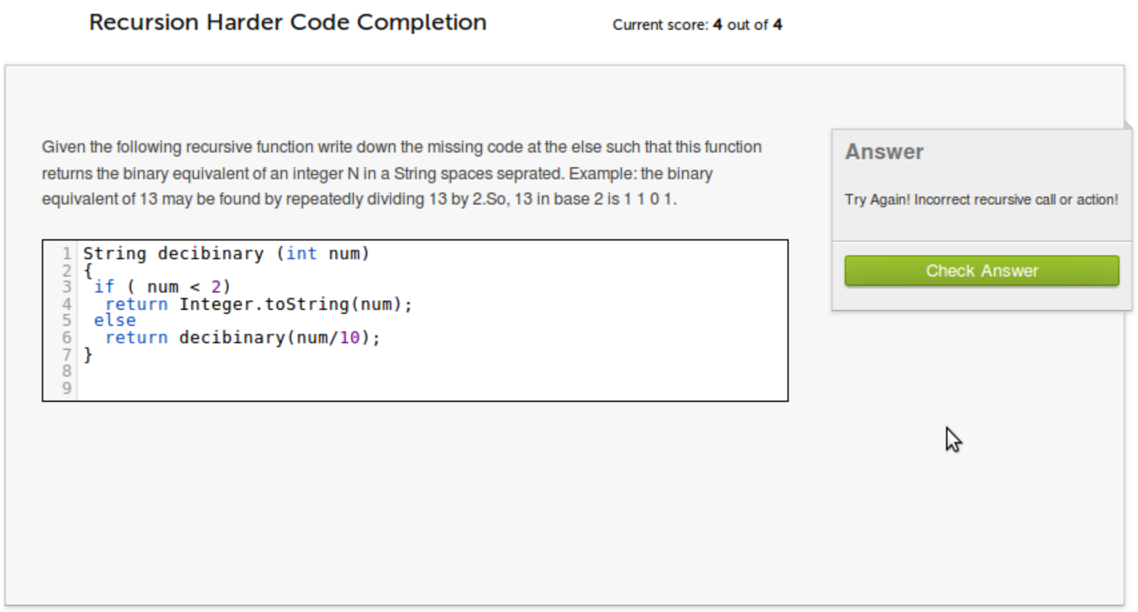
4.2.3 Programming Exercises

There are many good programming exercise examples in the literature for teaching recursion. Appendix A shows a detailed description of the examples and programming exercises presented in RecurTutor. Tables A.1 and A.2 shows, for each exercise, a brief description, which module it is used in, the recursion type, the misconceptions covered, and the reference we got the idea for the exercise from (if applicable).

There are mainly two types of programming exercises: writing exercises and tracing exercises. Writing exercises have two types: code completion and writing a full function. The code completion exercises ask the student to write one or two lines to complete a given function (e.g. a base case or a recursive call). The full function writing exercises ask the student to write a whole function that performs a certain task.

Feedback for the writing exercises has three cases:

- Correct: When the answer is perfectly correct in that it matches the output from the model answer on the test cases. Figure 4.4 shows an example of the feedback on the correct answer.
- 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 lead to infinite recursion. The feedback message gives information about why the answer is incorrect. Figure 4.5 and 4.6 shows an example of the feedback on the incorrect answer.
- Syntax error: When there are syntax errors. A full listing of the errors generated by


Recursion Harder Code Completion

Current score: **4** out of **4**

Given the following recursive function write down the missing code at the else such that this function returns the binary equivalent of an integer N in a String spaces seprated. Example: the binary equivalent of 13 may be found by repeatedly dividing 13 by 2.So, 13 in base 2 is 1 1 0 1.

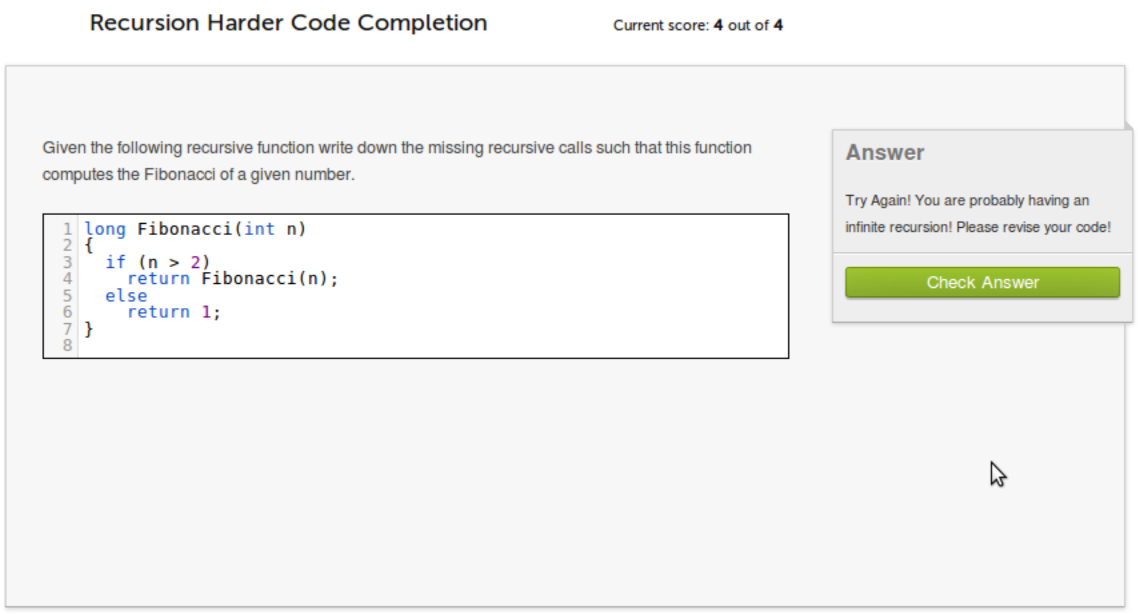
```
String decibinary (int num)
{
 if ( num < 2)
  return Integer.toString(num);
 else
  return decibinary(num/10);
}


```

Answer

Try Again! Incorrect recursive call or action!

Figure 4.5: Code completion programming exercise with feedback on the incorrect answer.


Recursion Harder Code Completion

Current score: **4** out of **4**

Given the following recursive function write down the missing recursive calls such that this function computes the Fibonacci of a given number.

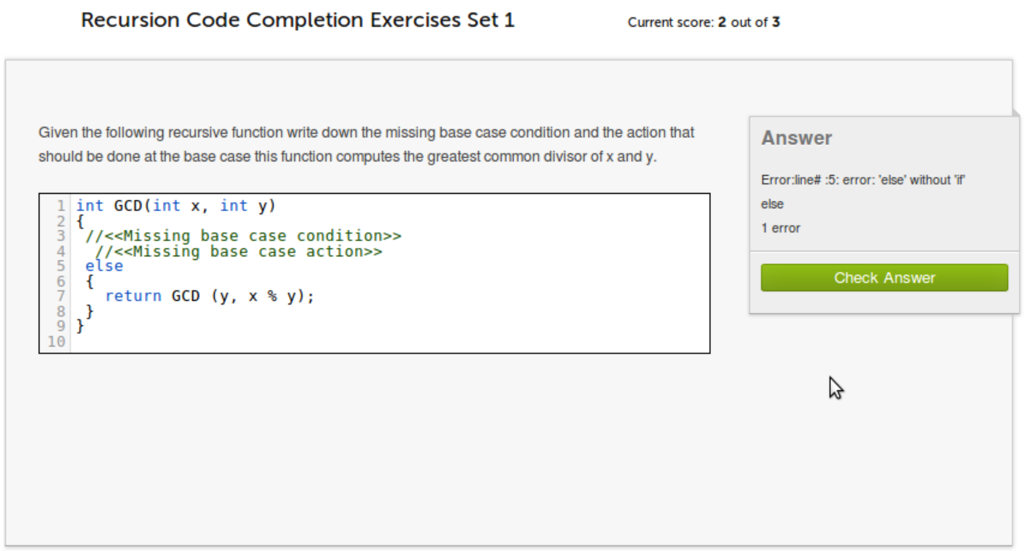
```
long Fibonacci(int n)
{
  if (n > 2)
    return Fibonacci(n);
  else
    return 1;
}

```

Answer

Try Again! You are probably having an infinite recursion! Please revise your code!

Figure 4.6: Code completion programming exercise with feedback on infinite recursion.



Recursion Code Completion Exercises Set 1

Current score: 2 out of 3

Given the following recursive function write down the missing base case condition and the action that should be done at the base case this function computes the greatest common divisor of x and y.

```
int GCD(int x, int y)
{
 //<<Missing base case condition>>
  //<<Missing base case action>>
 else
 {
   return GCD (y, x % y);
 }
}

```

Answer

Error:line# :5: error: 'else' without 'if'
else
1 error

Figure 4.7: Code completion programming exercise with feedback on syntax error.

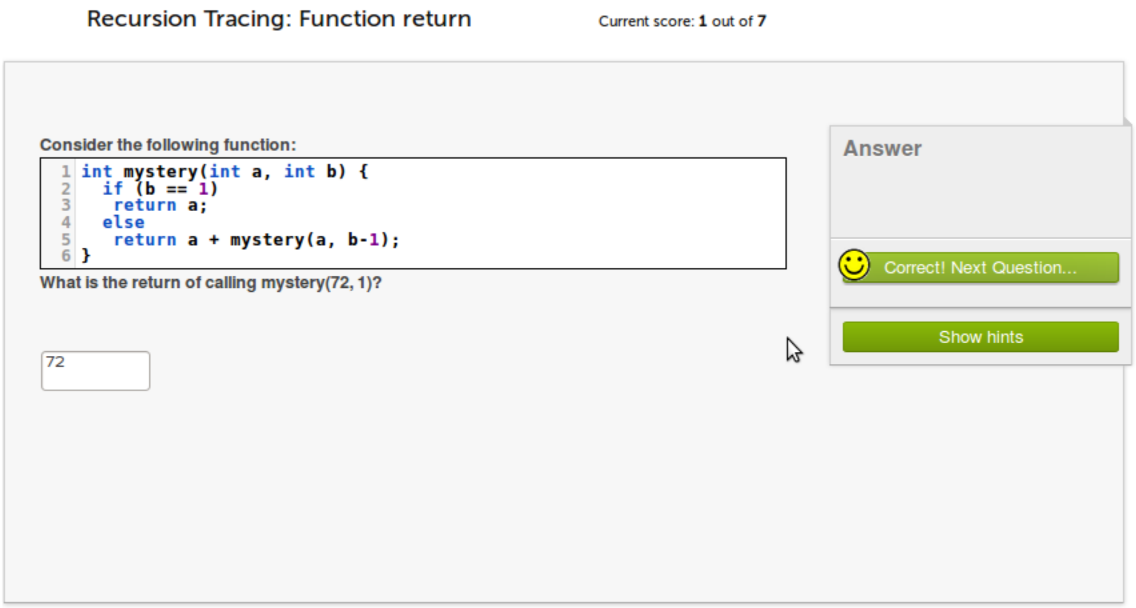
the compiler is shown to the student. Figure 4.7 shows the feedback on the answer with a syntax error.

The feedback for the tracing exercises is either correct or incorrect. The feedback is correct when the choice or the value entered is correct as shown in Figure 4.8 and incorrect in the other case as shown in Figure 4.9.

The tracing exercises provides hints when the student clicks on the “Show hints” button as shown in Figure 4.9. The hint appears below the code area. The tracing exercises provide randomized variables to change the presentation every time a student solves the problem. For example, the number 72 shown in the problem statement in Figure 4.8 is randomly generated.

4.3 Tutorial Infrastructure

This section briefly describes the RecurTutor programming exercise infrastructure. A full description for the latest OpenDSA infrastructure can be found in Chapter 3 in [24]. Our main contribution to the OpenDSA infrastructure is building and integrating the programming exercises automated assessment infrastructure. RecurTutor uses the existing OpenDSA infrastructure to create the prose, visualizations and non-programming exercises.


Recursion Tracing: Function return

Current score: 1 out of 7

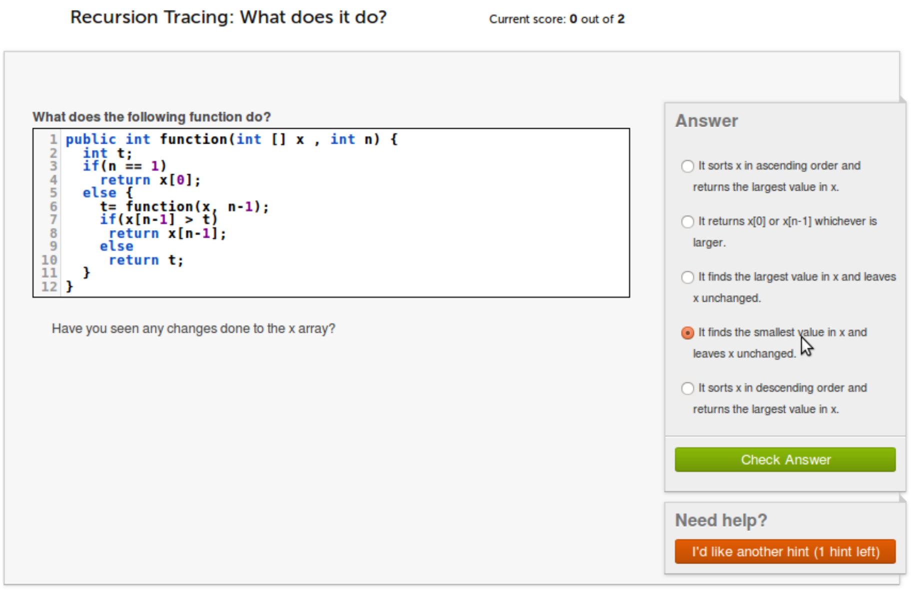
Consider the following function:

```
int mystery(int a, int b) {
  if (b == 1)
    return a;
  else
    return a + mystery(a, b-1);
}
```

What is the return of calling mystery(72, 1)?

72

Figure 4.8: Fill in blanks tracing exercise.


Recursion Tracing: What does it do?

Current score: 0 out of 2

What does the following function do?

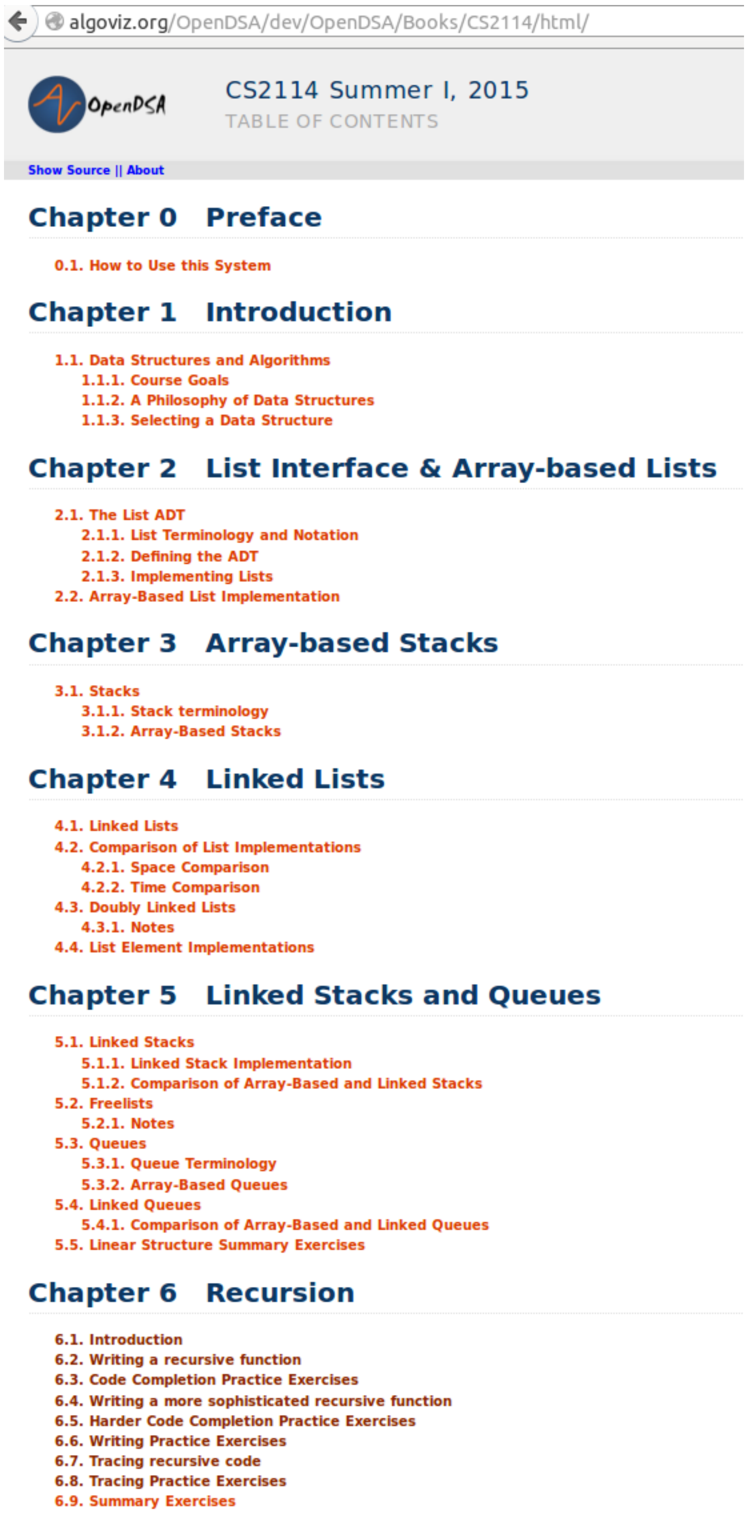
```
public int function(int [] x , int n) {
  int t;
  if(n == 1)
    return x[0];
  else {
    t= function(x, n-1);
    if(x[n-1] > t)
     return x[n-1];
    else
     return t;
  }
}
```

Have you seen any changes done to the x array?

Answer

- It sorts x in ascending order and returns the largest value in x.
- It returns x[0] or x[n-1] whichever is larger.
- It finds the largest value in x and leaves x unchanged.
- [✓ It finds the smallest value in x and leaves x unchanged.]
- It sorts x in descending order and returns the largest value in x.

Figure 4.9: Multiple choice tracing exercise.



algoviz.org/OpenDSA/dev/OpenDSA/Books/CS2114/html/

CS2114 Summer I, 2015

TABLE OF CONTENTS

Show Source || About

Chapter 0 Preface

0.1. How to Use this System

Chapter 1 Introduction

1.1. Data Structures and Algorithms
  1.1.1. Course Goals
  1.1.2. A Philosophy of Data Structures
  1.1.3. Selecting a Data Structure

Chapter 2 List Interface & Array-based Lists

2.1. The List ADT
  2.1.1. List Terminology and Notation
  2.1.2. Defining the ADT
  2.1.3. Implementing Lists
2.2. Array-Based List Implementation

Chapter 3 Array-based Stacks

3.1. Stacks
  3.1.1. Stack terminology
  3.1.2. Array-Based Stacks

Chapter 4 Linked Lists

4.1. Linked Lists
4.2. Comparison of List Implementations
  4.2.1. Space Comparison
  4.2.2. Time Comparison
4.3. Doubly Linked Lists
  4.3.1. Notes
4.4. List Element Implementations

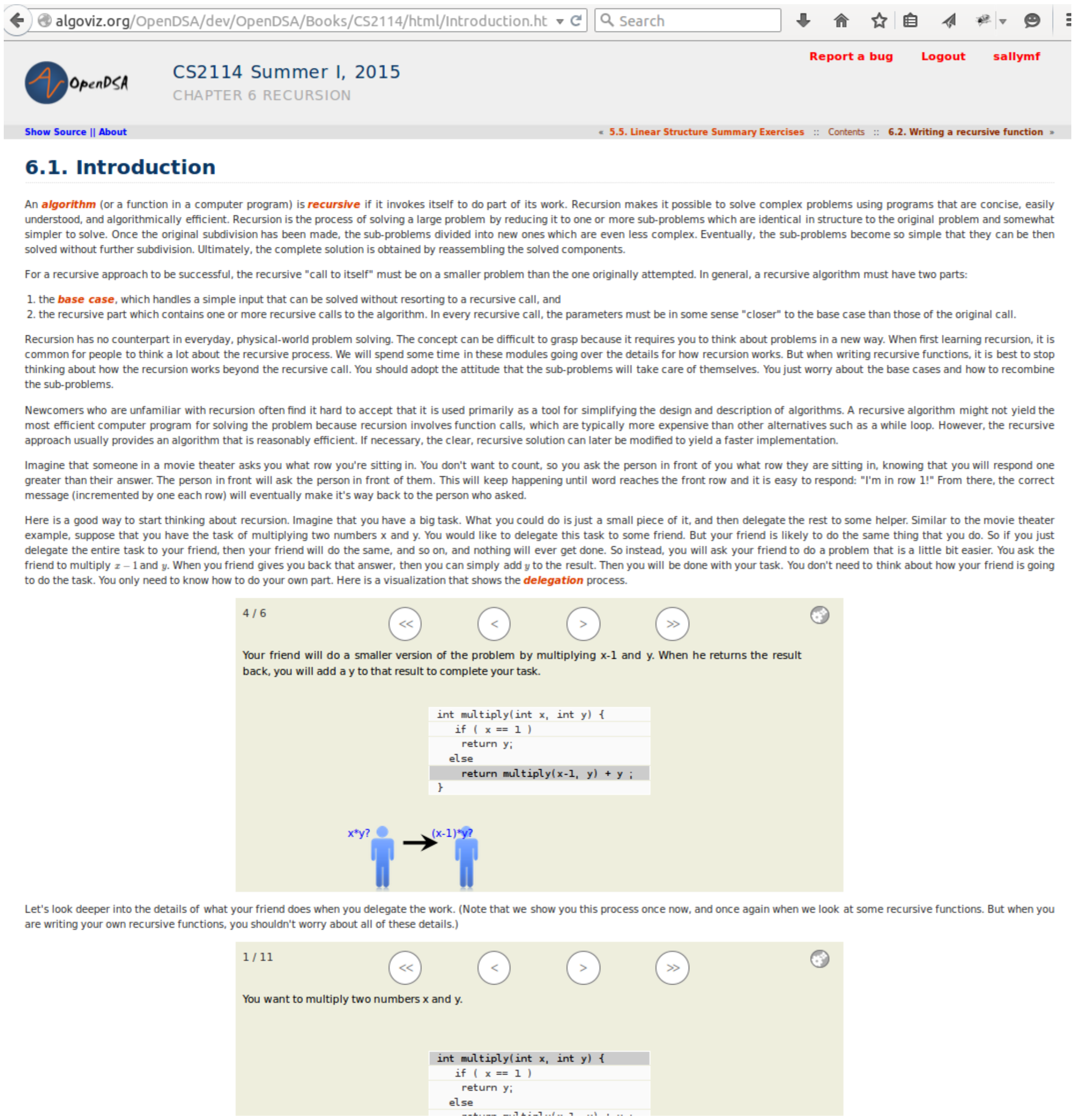
Chapter 5 Linked Stacks and Queues

5.1. Linked Stacks
  5.1.1. Linked Stack Implementation
  5.1.2. Comparison of Array-Based and Linked Stacks
5.2. Freelists
  5.2.1. Notes
5.3. Queues
  5.3.1. Queue Terminology
  5.3.2. Array-Based Queues
5.4. Linked Queues
  5.4.1. Comparison of Array-Based and Linked Queues
5.5. Linear Structure Summary Exercises

Chapter 6 Recursion

6.1. Introduction
6.2. Writing a recursive function
6.3. Code Completion Practice Exercises
6.4. Writing a more sophisticated recursive function
6.5. Harder Code Completion Practice Exercises
6.6. Writing Practice Exercises
6.7. Tracing recursive code
6.8. Tracing Practice Exercises
6.9. Summary Exercises

Figure 4.10: RecurTutor in the CS2114 book



algoviz.org/OpenDSA/dev/OpenDSA/Books/CS2114/html/Introduction.ht

Search

Report a bug   Logout   sallymf

CS2114 Summer I, 2015

CHAPTER 6 RECURSION

Show Source || About

« 5.5. Linear Structure Summary Exercises :: Contents :: 6.2. Writing a recursive function »

6.1. Introduction

An ***algorithm*** (or a function in a computer program) is ***recursive*** if it invokes itself to do part of its work. Recursion makes it possible to solve complex problems using programs that are concise, easily understood, and algorithmically efficient. Recursion is the process of solving a large problem by reducing it to one or more sub-problems which are identical in structure to the original problem and somewhat simpler to solve. Once the original subdivision has been made, the sub-problems divided into new ones which are even less complex. Eventually, the sub-problems become so simple that they can be then solved without further subdivision. Ultimately, the complete solution is obtained by reassembling the solved components.

For a recursive approach to be successful, the recursive "call to itself" must be on a smaller problem than the one originally attempted. In general, a recursive algorithm must have two parts:

1. the ***base case***, which handles a simple input that can be solved without resorting to a recursive call, and
2. the recursive part which contains one or more recursive calls to the algorithm. In every recursive call, the parameters must be in some sense "closer" to the base case than those of the original call.

Recursion has no counterpart in everyday, physical-world problem solving. The concept can be difficult to grasp because it requires you to think about problems in a new way. When first learning recursion, it is common for people to think a lot about the recursive process. We will spend some time in these modules going over the details for how recursion works. But when writing recursive functions, it is best to stop thinking about how the recursion works beyond the recursive call. You should adopt the attitude that the sub-problems will take care of themselves. You just worry about the base cases and how to recombine the sub-problems.

Newcomers who are unfamiliar with recursion often find it hard to accept that it is used primarily as a tool for simplifying the design and description of algorithms. A recursive algorithm might not yield the most efficient computer program for solving the problem because recursion involves function calls, which are typically more expensive than other alternatives such as a while loop. However, the recursive approach usually provides an algorithm that is reasonably efficient. If necessary, the clear, recursive solution can later be modified to yield a faster implementation.

Imagine that someone in a movie theater asks you what row you're sitting in. You don't want to count, so you ask the person in front of you what row they are sitting in, knowing that you will respond one greater than their answer. The person in front will ask the person in front of them. This will keep happening until word reaches the front row and it is easy to respond: "I'm in row 1!" From there, the correct message (incremented by one each row) will eventually make it's way back to the person who asked.

Here is a good way to start thinking about recursion. Imagine that you have a big task. What you could do is just a small piece of it, and then delegate the rest to some helper. Similar to the movie theater example, suppose that you have the task of multiplying two numbers x and y. You would like to delegate this task to some friend. But your friend is likely to do the same thing that you do. So if you just delegate the entire task to your friend, then your friend will do the same, and so on, and nothing will ever get done. So instead, you will ask your friend to do a problem that is a little bit easier. You ask the friend to multiply $x-1$ and y. When you friend gives you back that answer, then you can simply add y to the result. Then you will be done with your task. You don't need to think about how your friend is going to do the task. You only need to know how to do your own part. Here is a visualization that shows the ***delegation*** process.

4 / 6

Your friend will do a smaller version of the problem by multiplying x-1 and y. When he returns the result back, you will add a y to that result to complete your task.

```
int multiply(int x, int y) {
   if ( x == 1 )
     return y;
  else
     return multiply(x-1, y) + y ;
}
```

Let's look deeper into the details of what your friend does when you delegate the work. (Note that we show you this process once now, and once again when we look at some recursive functions. But when you are writing your own recursive functions, you shouldn't worry about all of these details.)

1 / 11

You want to multiply two numbers x and y.

```
int multiply(int x, int y) {
   if ( x == 1 )
     return y;
  else
```

Figure 4.11: An example of a lesson in RecurTutor

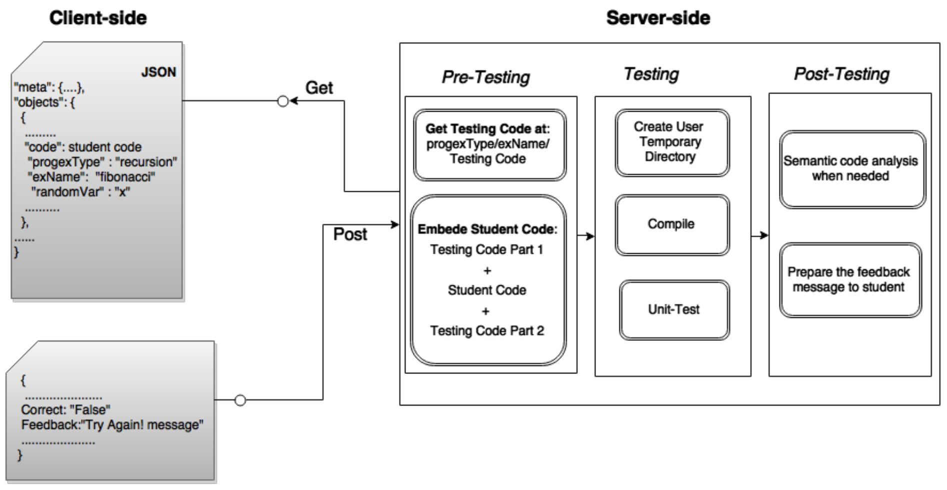

Client-side
Server-side
JSON
"meta": {....},
"objects": {
{
....
"code": student code
"progexType" : "recursion"
"exName": "fibonacci"
"randomVar" : "x"
....
},
......
}
Get
Post
{
....
Correct: "False"
Feedback:"Try Again! message"
....
}
Pre-Testing
Testing
Post-Testing
Get Testing Code at: progexType/exName/ Testing Code
Embede Student Code: Testing Code Part 1 + Student Code + Testing Code Part 2
Create User Temporary Directory
Compile
Unit-Test
Semantic code analysis when needed
Prepare the feedback message to student

Figure 4.12: Communication between the client and the server for the programming exercises

4.3.1 Client-side Infrastructure

Since we use the OpenDSA infrastructure [24], the client-side interface for the programming exercises is in HTML5.

We use Codemirror * to provide the code coloring and the line numbering for the Java code in the writing and the tracing exercises.

We make use of HTML5's localStorage feature to keep all transactions in a coherent state on the client.

For the writing exercises, the interface is typically a function signature and an editor in which the student writes or pastes his code. The student inputs his code then clicks on the "Check Answer" button. The student code is then sent to an evaluation process running on the server, after which feedback is returned to the student. Figures 4.4, 4.5, 4.6 and 4.7 show examples of the interface shown to the student for writing programming exercises. Figures 4.8 and 4.9 show examples for two tracing exercises.

The tracing exercises are evaluated on the client through JavaScript code. The writing exercises are evaluated on the server using unit testing.

4.3.2 Automated assessment on the server

As shown in Figure 4.12, output-based assessment uses a test suite to test the student's code. The testing process has three phases: pret-testing, testing (unit-test and semantic), and post-testing. In the pre-testing phase, the student code is embedded into a test harness

*https://codemirror.net/

(since it is only a function or few lines of code and must be part of a complete program to execute) and the code is compiled. We create a temporary directory for each submission that has all of the temporary files that may be needed during the code testing. In the testing phase, the testing main process runs the unit test cases, which are designed to call the student code and get back the output of the code, and then checks if the output is correct or not. We run the student code in a separate thread, which times out if it is taking too long time to run (typically around 2 seconds). In the post-testing phase, feedback is sent back to the client which displays it to the student.

We have resolved some technical issues like the problem of copying and pasting code from an external editor or text in an email to the code editor in the browser. The pasted code might contain non-printable characters which might cause run-time errors. This issue is resolved by cleaning the student's code of any non printable characters before running the code for assessment, so that the student can copy and paste the code to the online editor with no problems. The student's code is sand boxed using a Java security policy file so that the student is not able to run malicious code nor read or write files on the system through the code editor.

In Chapter 7, we discuss the semantic code analysis shown in the post-testing phase in Figure 4.12 for the recursion in binary tree tutorial.

4.4 Summary

This chapter presented RecurTutor tutorial content and infrastructure. The contributions presented in the RecurTutor system are:

- A practice-based approach as a new way to teach recursion that can be used for other hard to understand topics in computer science courses.
- A complete tutorial that uses textual content, visualizations and programming exercises to teach hard topic like recursion.
- Immediate feedback on the automatically assessed exercises based on the student's answer.
- Hints to help students answer the exercises.

Chapter 5

RecurTutor's Impact on Students

In this chapter, we show impacts of using RecurTutor. We start this chapter by showing the results of surveys on the time students spend on recursion and their confidence level when using RecurTutor. We then compare time spent by the students who have not used the recursion tutorial (control group) versus the students who have used it (experimental group). That will answer the following research questions:

Does using our tutorial lead to students spending time in line with what instructors believe to be appropriate for learning and understanding basic recursion?

We show the effect of using the tutorial on learning outcomes (exam scores in our case). The analysis of these results will answer the following research question:

Does using the basic recursion tutorial support basic recursion learning than typical instruction?

We show the results of a questionnaire that the students completed about the experience of using RecurTutor. This will answer the following research question:

Does using the basic recursion tutorial enhance the confidence level of the students on this topic?

Last, we show an analysis of the logs for students who used RecurTutor to understand student behaviors when using the tutorial and whether those behaviors related to student performance. Estimates of time spent on the tutorial derived from log analysis were a reasonable match to self-reported times in the surveys. In addition, log analysis shows that the performance on the RecurTutor exercises support exam performance.

5.1 The Impact of RecurTutor

5.1.1 Confidence Level and Time Spent on Recursion

In this section, we show the results gathered from student surveys during Spring 2015. Then we compare these results to the results of the Spring 2014 surveys to show the effect of using RecurTutor on students' confidence level and time spent on recursion.

Participants: The participants were students enrolled in CS2114 Data Structures and Software Design during Spring 2015 at Virginia Tech. CS2114 is a programming-intensive class with two hours of programming labs each week. OpenDSA exercises were used as weekly mandatory homework assignments. RecurTutor was assigned on three of those assignments. Many students also practiced (voluntarily) recursion using RecurTutor to prepare for midterms and final exam.

Materials and procedure: During the last lab session for CS 2114, students were given a paper survey regarding their experience with learning recursion. A total of 83 students completed 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 spent on solving the OpenDSA exercises on recursion?
3. How many hours did you spend reading the OpenDSA recursion chapter (not counting the time spent on solving the exercises)?
4. On a scale of 1-5, rate your confidence level about your mastery of recursion. (1 being least confident to 5 being most confident)

Results: The key results are shown in Table 5.1.

Table 5.1: Spring 2015 students survey responses on time on recursion

Question	**Mean**
Time on Recursion out-of-class	7.3 hrs
Time reading RecurTutor	1.65 hrs
Time on RecurTutor Exercises	3.3 hrs
Confidence level	3.06

In order 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. The results of the t-test comparing the time spent on recursion for Spring 2015 versus Spring 2014 are shown in Table 3.2. Table 5.2 shows the results of the t-test:

Table 5.2: A t-test comparing the time spent on recursion for Spring 2015 versus Spring 2014

	Sp15(N=83)		Sp14(N=54)		p-value
	mean	std dev.	mean	std dev.	
Time on Recursion (hrs)	7.3	7.4	4	4.1	0.1385
Time on Prog Ex (hrs)	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 the tutor is significantly more.

5.1.2 Exam Scores

In Spring 2015, before students were introduced to recursion in class, they were given a recursion pre-test in order to know their previous knowledge of recursion. After being introduced to recursion in class and using RecurTutor, they were give recursion questions on the final exam. We measure the relative performance by comparing the post-test (exam) scores for the students who did not use the tutorial (the control group) versus who did use it (the experimental group).

Pre-test

Participants: The participants were students enrolled in CS2114 Data Structures and Software Design course during Spring 2015 ($n = 166$) at Virginia Tech. The participants had not been yet introduced to recursion in class, nor used RecurTutor.

Materials and procedure: During Spring 2015, the participants were given 5 questions on recursion in an in-class pre-test in the CS2114 course. The questions can be found in Appendix C.1. The students were aware that they were receiving a participation grade.

Results: All the questions scores were rescaled to be out of 100. The means are shown in Table 5.3.

Table 5.3: Means of the recursion pre-test questions in Spring 2015

Question	**Mean**
Infinite Recursion	91.5
Code Tracing	29.5
Code Tracing Multiple Recursive calls	17.5
Code Writing	20.0
Code Writing Multiple Recursive calls	4.0

We see from the means that, excluding the infinite recursion question, the majority of the students do not have enough background knowledge on recursion to answer a recursive tracing or writing question.

Post-test

Participants: The participants were 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. The participants had used RecurTutor.

We compared the final exam scores of students in two sections that did not use the tutorial (the control groups, $n = 215$ and $n = 157$) and a section that did (the experimental group $n = 168$).

Materials and procedure: During Spring 2015, the participants were given 4 questions on recursion on the final exam of the CS2114 course. The questions can be found in Section C.1 in Appendix C. During Fall 2014, the same questions were given to the students. During Spring 2014, the same questions were given to the students except for the writing question. Using an unpaired t-test ($\alpha = 0.05$), we have compared the students scores on each recursion question between the following pairs: Fall 2014 versus Spring 2015, Spring 2014 versus Spring 2015, and Fall 2014 versus Spring 2014.

Results: All of the question scores were rescaled to be out of 100. Tables 5.4, 5.5, and 5.6 show the results of the t-tests for each questions for Fall 2014 versus Spring 2015, Spring 2014 versus Spring 2015, and Spring 2014 vs. Fall 2014, respectively.

Table 5.4: t-tests for recursion question exam scores for Fall 2014 versus Spring 2015

	F14(N=215)		Sp15(N=168)		p-value
	mean	std dev.	mean	std dev.	
Writing	59.7	30.0	69.7	21.0	0.0003*
Tracing	93.99	22.34	98.36	11.36	0.0219*
Infinite Recursion	97.21	16.5	99.40	7.78	0.0885
Code Completion	83.25	37.5	82.74	37.91	0.8940

* = statistically significant

Table 5.5: t-tests for recursion questions exam scores for Spring 2014 versus Spring 2015

	Sp14(N=157)		Sp15(N=168)		p-value
	mean	std dev.	mean	std dev.	
Tracing	88.64	30.20	98.36	11.36	0.0001*
Infinite Recursion	95.45	20.73	99.40	7.78	0.0227*
Code Completion	70.7	45.31	82.74	37.91	0.0037*

* = statistically significant

Table 5.6: t-tests for recursion questions exam scores for Spring 2014 versus Fall 2014

	Sp14(N=157)		F14(N=215)		p-value
	mean	std dev.	mean	std dev.	
Tracing	88.64	30.20	93.99	22.34	0.0503
Infinite Recursion	95.45	20.73	97.21	16.51	0.3591
Code Completion	70.70	45.31	83.25	37.5	0.0096*

* = statistically significant

Control versus Experimental Group Summary Results
An unpaired t-test ($\alpha = 0.05$) was used to compare the RecurTutor group to the combined control groups. The performance results are shown in Table 5.7. From the table, we find 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. These are considered moderate effect sizes.

Table 5.7: Control versus Experimental Group Summary Results

Question	**p-value**	**Effect Size**	**Control Mean**	**Experimental 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

The findings from the t-tests 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 is not a good question to measure students performance on recursion as it does not match well the misconceptions we had for basic recursion and we see that even without the tutorial that question's scores vary significantly between Spring 2014 and Fall 2014 semesters.
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. We interpret this as support for the hypothesis that using RecurTutor was the reason for the improved scores.

5.2 Treatment differences between the control and the experimental group

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 requires future investigation. We are trying to answer the following research question:

What are the differences between the control and the experimental treatments that could have contributed to the performance and non-performance enhancement of the experimental group?

1. Differences between the CodingBat exercises and RecurTutor exercises: Table 5.8 shows

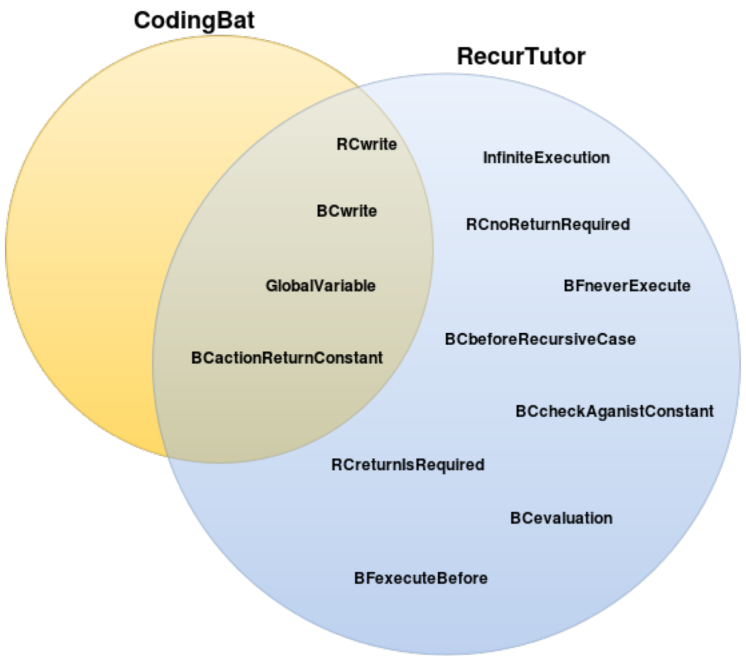


Figure 5.1: Misconceptions covered by CodingBat and RecurTutor

the main differences between CodingBat recursion exercises solved by the control group and RecurTutor exercises solved by the experimental group. Figure 5.1 shows the misconceptions covered by both CodingBat and RecurTutor. We see from the figure that CodingBat only covers 30% of the misconceptions 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.

Table 5.8: 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

5.3 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). The purpose of doing item analysis is to know if the questions that we used can correctly predict student ability on recursion.

We have used the ltm* R package to perform the item analysis. We used the two-parameter logistic model which takes into consideration the discrimination and the difficulty. Table 5.9 shows the difficulty and discrimination indices computed by ltm.

Table 5.9: 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

We mapped the discrimination index computed by ltm to percentages (as the percentages 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 well with the misconceptions that we have identified on basic recursion. For these reasons we exclude it from further consideration.

The student ability measures are the scores of theses 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 $\alpha > .9$, which indicates a highly reliable test for 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

*https://cran.r-project.org/web/packages/ltm/

question (item) quality by constructing item response curves (IRCs) using ltm package. The IRCs for the three exam questions are shown in Figure 5.2. The IRC demonstrates the desired correlation between conceptual knowledge and item performance for the three items. So as the student ability increases, the probability to solve 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.

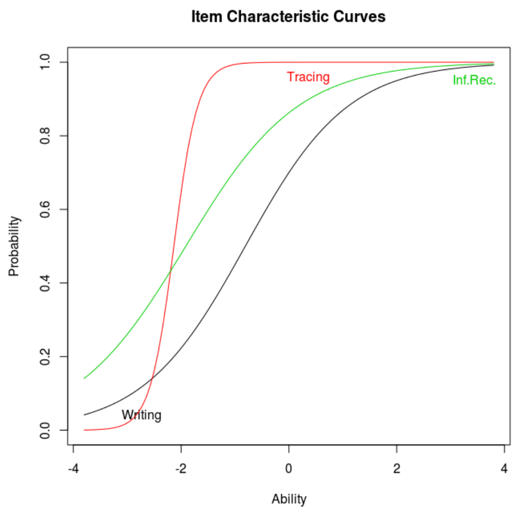


Figure 5.2: Item response curves for the questions used to measure student performance on recursion

5.4 Student opinions on RecurTutor

Participants: The participants were students enrolled in CS2114 Data Structures and Software Design course during Spring 2015 at Virginia Tech. The participants had used RecurTutor.

Materials and procedure: During the last lab session of CS 2114, students were given a paper survey regarding their opinion on RecurTutor. A total of 83 students filled in the survey and returned it back to the teaching assistant at the end of the lab. The questions were as follows:

1. Have you used any materials other than OpenDSA to learn recursion? if yes, then what materials?
2. On a scale of 1-5, rate OpenDSA's online recursion chapter? (1 is poor to 5 is excellent)
3. Please describe your overall opinion of your experience with the OpenDSA recursion tutorial.

Results

Use of other materials: Table 5.10 shows the percentage of the students who have used resources other than RecurTutor.

Table 5.10: Students use of materials other than RecurTutor

Material	**Percentage**
Google	13.3%
Text Book	13.3%
Youtube	8.4%
StackOverFlow	7.2%
Coding bat	6.0%
Wikipedia	2.4%
Wolfram	1.2%
Unspecified	3.6%
None	21.7%

RecurTutor rating Table 5.11 shows the analysis of student ratings for RecurTutor.

Table 5.11: Students ratings of RecurTutor

Parameter	**Value**
Mean	3.4
Standard Deviation	0.8%
Standard error of the mean	0.1%
90% Confidence Interval	3.2 to 3.5
95% Confidence Interval	3.2 to 3.6
99% Confidence Interval	3.1 to 3.6
Minimum	1
Maximum	5
Number of Students	83

Overall Opinion

We use the same codes used by Fouh [24] to label ideas expressed in each response. The codes were used to capture students opinion of RecurTutor, and also elements that influenced their opinion. Table 5.12 shows a description for each code (label).

Table 5.13 shows the percentage of the opinions of students per each opinion code. The free response question asking about the overall opinion was answered by 79 students out of 83.

Table 5.12: Student Opinion Coding scheme description [24]

Label	**Description**
Positive	Positive experience with no details
Positive-interactivity	Positive experience- RecurTutor interactive elements
Positive-frustration	Positive experience - RecurTutor limitations or defects
Neutral	Neither good or bad experience
Neutral-frustration	Neither good or bad experience - RecurTutor limitations or defects
Negative-frustration	Negative experience - RecurTutor limitations or defects

Table 5.13: Percentage of the opinions of the students per each opinion code.

Label	**Percentage**	
Positive	20.25%	64.55% Positive
Positive-interactivity	15.19%	
Positive-frustration	29.11%	
Neutral	6.33%	20.25% Neutral
Neutral-frustration	13.92%	
Negative-frustration	15.18%	15.18% Negative

The frequently reported frustrations that students reported were all about the programming exercises:

1. There is a jump in difficulty in some for the programming exercises. (16%)
2. The instructions are not clear enough for some exercises. (10%)
3. Would like to save the code to view it later in case it is correct. (8%)
4. The feedback (on incorrect submissions) is not sufficient for some exercises. (7%)

5.5 Student use of RecurTutor

This section discusses analysis of the student interaction logs taken while using RecurTutor during Spring 2015.

5.5.1 Proficiency Seekers

In this section we discuss the following question:

Do students behave in a certain way to get credit that they should not get? We name such students "Proficiency Seekers".

RecurTutor includes a substantial number of programming exercises. A few of these exercises are relatively difficult for many of the students. As a result, many students are motivated to seek ways to get around doing those particular exercises.

As detailed in Chapter 4, the programming exercises fall into the following types: code completion, hard code completion, writing, and tracing exercises. RecurTutor has at least one set of exercises for each type. For each set, for a student to get proficiency (credit), he must solve a certain number of exercises. In each set, once a student solves an exercise correctly, he is given another randomly picked exercise to solve.

The exercises are implemented based on the Khan Academy Exercise Framework. A given "exercise" is actually several specific programming problems, of which the students are supposed to do a random subset, say two or three out of five. Students can see the exercise that comes up at random, then reload the page without penalty to get another exercise at random. As a result, some students may do one exercise and getting it correct, then simply keep reloading the page until that exercise repeats, then do it again. So if an exercise requires the student to solve three programming problems to get credit, a student can game the system by solving the first problem correctly, then when the system presents the next problem, the student can just keep reloading the page to get the first problem selected randomly one more time. They then solve it again, and repeat this behavior to get the same problem solved three times but counted as three problems.

We have analyzed student responses to the programming exercises to find out how many students did repeat the same exercise in consecutive attempts to earn credit. (We define this as students who solved the same programming exercise twice or more in consecutive attempts, in less than 30 minutes, get them all right, and have used the same answer for all of the consecutive attempts.) We have found the following.

1. 52% of students who attempt the programming exercises do—on at least one instance—exhibit this behavior of reloading the page to get the same problem that they have just solved correctly, then re-solve it to get a second credit instead of solving another one.
2. According to the student surveys, the students consider the exercises that ask them to write a full function or complete a non-simple recursive function which has more than one recursive call or base case as hard. 93% of the "Proficiency Seekers" skipped exercises that are considered hard. We have not found that students exhibit this behavior for simple code completion or the tracing exercises.
3. 42% of "Proficiency Seekers" have repeated this behavior for all three recursive function writing exercise sets.

Table 5.14 shows the percentage of students who gamed each set of the writing programming exercises. As mentioned before students did not appear to game the tracing exercises.

As a result of this analysis, we are re-designing the tutorial and we will revise the framework infrastructure to avoid the negative behavior. In the revised version of the tutorial, we removed certain hard exercises that more than 80% of the students avoided. We agree that the very hard exercises in the original tutorial were out of line with the appropriate

Table 5.14: For each writing exercise, the percentages of "Proficiency Seekers" or students who avoided writing programming exercises by repeatedly reloading the page to repeat an exercise.

Group	**Percentage**
Code Completion Set 1	0%
Code Completion Set 2	0%
Code Completion Set 3	0%
Code Completion Set 4	0%
Harder Code Completion	31%
Writing Set 1	67%
Writing Set 2	72%
Writing Set 3	80%

difficulty level. We have also presented the remaining writing exercises individually, instead of grouping them into sets. The net result is that nearly all students in future will do more exercises and of more appropriate difficulty.

5.5.2 Visualization Skimmers

In this section, we discuss the following question:
Do students go through the visualizations and read then thoroughly, or do they skim over the slides until they reach the end of the visualization? We name students who skim over the slides until they reach the end of the visualization "Visualizations Skimmers"

To find an estimate of the appropriate amount of time that should be spent on each visualization, we have asked one undergraduate and three graduate students from the OpenDSA team to voluntary go through the visualizations, read and understand each of them, then provide us with the time spent on each visualization. We then averaged their reported times. Table 5.15 shows the name of each visualization in the order that it appears in the tutorial, and the averaged reported times in seconds that should be spent by the student to understand it. We have computed the accumulated time spent by each student on each visualization through out the whole semester. In other words, for each visualization, we have summed the time each student spent on this visualization along the whole semester. Table 5.16 shows the percentage of the students who spent less than half and the percentage of the students who spent less than quarter of the averaged reported times for each visualization.

Table 5.15: Estimated time to read and understand each visualization

Name	**# of Slides**	**Averaged time (secs)**
Introduction Delegation	6	85
Introduction Detailed	11	93
Writing Steps	22	160
Writing Sum	8	96
Tracing Winding	17	80
Tracing Sum	13	80
Tracing Factorial	11	81
Tracing Sum 2	4	40
Tracing Domino	6	90
Tracing Domino Count	5	60
Tracing Domino Print	3	60
Tracing TOH	141	280

Table 5.16: Percentage of students who spent less than half and quarter of the average time shown in Table 5.15 for each visualization

Name	**Percentage $< 1/2$**	**Percentage $< 1/4$**
Introduction Delegation	78.3%	50.6%
Introduction Detailed	72.0%	40.1%
Writing Steps	41.5%	36.8%
Writing Sum	57.3%	23.0%
Tracing Winding	85.5%	52.6%
Tracing Sum	65.7%	54.6%
Tracing Factorial	85.5%	67.1%
Tracing Sum 2	86.2%	76.3%
Tracing Domino	88.8%	74.3%
Tracing Domino Count	82.2%	73.0%
Tracing Domino Print	97.4%	92.6%
Tracing TOH	68.4%	27.6%

5.5.3 Gaming and skimming behaviors versus student performance

In this section, we relate log data to final exam scores to see if there are different patterns of performance based on proficiency seeking and visualization skimming. We will try to answer the following questions:

- **Is there a correlation between student performance on exams and proficiency seeking behavior (i.e., gaming the exercises by refreshing the page to skip a programming exercise)?**

- **Is there a correlation between student performance on exams and visualization skimming?**

We grouped students into quartiles by final exam score. In each of the quartiles, for each student, we have looked at the log analysis to see if the student exhibited proficiency seeking gaming behavior done at least once, twice, three, or four times. Quartile 1 represents students who performed below the 25^{th} percentile on the exam; Quartile 2 represents students with scores between the 25^{th} and the 50^{th} percentile; Quartile 3 represents students with scores between the 50^{th} and the 75^{th} percentile; and Quartile 4 represents students with scores above the 75^{th} percentile. Table 5.17 shows the statistics for the different groups.

Table 5.17: Percentage of students who gamed the programming exercises at least 1, 2, 3, and 4 times per each quartile

Group	**1 time**	**2 times**	**3 times**	**4 times**
Quartile 1(N=44)	47.6%	28%	18.6%	14%
Quartile 2(N=43)	54.1%	23.3%	16.3%	11.6%
Quartile 3(N=43)	47.5%	20.9%	14%	4.6%
Quartile 4(N=43)	44%	14%	4.6%	0%

We have done an ANOVA to see if exam score predicts the gaming behavior. Table 5.18 shows that the R-square does not indicate that the exam score can predict the gaming behavior. The F Ration also indicates a non-significance, which means that student score does not predict the gaming behavior.

Table 5.18: ANOVA to see if exam score predicts the gaming behavior

Rsquare	**F Ratio**	**Prob > F**
0.032	1.87	0.136

We have done a multivariate ANOVA (MANOVA) to see if students from different quartiles behave differently in terms of the number of programming exercises gamed. Table 5.19 shows the MANOVA results. The MANOVA shows that low-performing students gamed the programming exercises more than did high-performing students. However, the ANOVA analysis of Table 5.18 showed that we cannot predict student gaming behavior from their exam score. The MANOVA analysis of Table 5.19 groups student into quartiles based on their final exam scores, while the ANOVA analysis of Table 5.18 compares raw exam scores against gaming behavior. The MANOVA results showed that when comparing students of similar ability level (i.e., students who belong to the same quartile) without taking into consideration the individual differences of students within the quartile, the means for the number of times that students game the programming exercises varies significantly between quartiles, with low-performing students gaming more. In contrast, the ANOVA showed that we cannot predict the number of times that a student gamed the programming exercises based on their final exam score. Combining the two analyses, we know that low performing students game more, but we can not actually predict how much a student games programming exercises from their final exam score.

Table 5.19: MANOVA comparing how students from different quartiles behave differently in terms of the number of programming exercises gamed where students grouped by final exam scores

Test	**Value**	**F**	**Prob > F**
F Test	0.583	97.4	<0.0001*

* = statistically significant

We have done a Quartile analysis to determine if the behavior of skimming over the visualizations is related to how well a student performed. For this analysis, we consider a student to be skimming over a visualization if he spent less than half of the estimated adequate time for this visualization. We then compute the number of the visualizations the students have repeated this behavior. If it is repeated for more than 75% of the visualization, which is 9 visualizations (out of 12) in our case, we considered the student as a visualization skimmer. We also computed the percentages for students who skimmed more than 50% and more than 25% of the visualizations. Table 5.20 shows the statistics for the different groups.

We performed t-tests between all pairs combinations of student groups. None of the t-tests showed a significant difference in the skimming behavior that can be interpreted by student ability.

Table 5.20: Percentage of students in each quartile who spent less than half of the reported average times more than 75% , 50% and 25% of the visualizations

Group	**> 75%**	**> 50%**	**> 25%**
Quartile 1 (N=44)	61.50%	87.18%	94.87%
Quartile 2 (N=43)	47.20%	72.22%	100.00%
Quartile 3 (N=43)	53.80%	79.49%	97.44%
Quartile 4 (N=44)	57.80%	76.32%	94.73%

We have repeated the same analysis but this time we consider a student to be skimming over a visualization if he spent less than quarter of the estimated adequate time for this visualization. We performed t-tests between all pairs combinations of student groups. None of the t-tests showed a significant difference in the skimming behavior that can be interpreted by student performance. Table 5.21 shows the statistics for the different groups.

As seen from the results, we found that proficiency seeking behavior was noticed more for students with lower ability which means that they game more the programming exercises.

Table 5.21: Percentage of students in each quartile who spent less than quarter of the reported average times more than 75% , 50% and 25% of the visualizations

Group	**> 75%**	**> 50%**	**> 25%**
Quartile 1 (N=44)	41.03%	64.10%	87.17%
Quartile 2 (N=43)	33.33%	50.00%	69.44%
Quartile 3 (N=43)	17.95%	56.41%	82.05%
Quartile 4 (N=44)	28.94%	55.26%	68.42%

On other hand, we could not find an evidence for any relationship between exam performance and visualization skimming behavior.

5.5.4 Time spent on RecuTutor

We have analyzed the student logs to know if the actual time spent by the students on the recursion tutorial matches the self-reported time given in the survey responses. We wanted to know the time that they spent on solving the programming exercises, time spent going through the visualizations, and the total time spent on the recursion tutorial. We discuss the following question:
Does the self-reported time spent on recursion out-of-class match the time computed from from the students logs?

Table 5.22: The time spent by the students on the programming exercises in minutes

Group	**Median**	**Mean**
Code Completion Set 1	22.93	42.12
Code Completion Set 2	17.68	28.3
Code Completion Set 3	21.2	28.88
Code Completion Set 4	21.275	30.96
Harder Code Completion	26.2	37.31
Writing Set 1	36.14	80.23
Writing Set 2	21.58	38.84
Writing Set 3	15.57	29.61
Code Tracing Set 1	20.77	52.92
Code Tracing Set 2	9.18	25.44
Code Tracing Set 3	13.04	35.03
Code Tracing Set 4	12.925	52.74
Code Tracing Set 5	15.083	35.31
Code Tracing Set 6	17.77	41.95

Table 5.22 shows the median and the mean of the time spent by students on the recur-

sion tutorial exercises. According to Table 5.22, the mean of the total time spent on the programming exercises is around 5 hours.

In addition, we have found from the logs that the median and the mean of the time spent on the non programming summary questions are 2.23 and 3.65 minutes respectively. The median and the mean of the time spent on the visualizations are 10 and 17 minutes respectively. So adding those numbers and taking into consideration the time spent on reading the text, the median total time spent on the recursion tutorial as conservatively calculated from the log data (i.e. underestimate) is around 6 hours. This is reasonably close to the 7.3 hours reported by the students on the surveys.

We have also done a log analysis to find out the total time spent by the students on all the recursion tutorial modules. For each page/module in the recursion chapter we have calculated the time spent between when the user opened the module until he closed the page or moved to the next one. In our analysis we have excluded any action that takes from the student more than half an hour. We exclude longer sessions because the likely explanation for these cases that the student opened the page and left his computer to do something else. This analysis shows that the mean of the total number of hours spent on RecurTutor is 10 hours, which is likely to be an over estimate. In summary, the 6 hours (underestimate) and 10 hours (overestimate) computed from the log data is a reasonable range around the self-reported time of 7.3 hours. This time analysis is important because it gives us confidence in both measures (since they support each other), and it gives us confidence in using these techniques in the future.

5.5.5 Item analysis for the tutorial exercises

The aim of doing item analysis for tutorial exercises is to understand better the quality of the tutorial exercises and how well they measures student skill at recursion. Item analysis requires a score for each student attempt. However, OpenDSA uses a mastery approach, where students typically repeat exercises until they gain credit. So we needed to find a way to assign each student a score on each exercise that he attempted. We mapped student submissions to actual scores the following way:

- Writing exercises:
    1. We excluded all the attempts that have syntax errors as we believe these have nothing to do with skill on recursion.
    2. We used as a performance indicator the time spent by a student until getting the exercises correct. This has the effect of accounting for the number of attempts, since more attempts requires more time. For all exercises, we find that the distributions of the time spent were skewed normal. Therefore we have done a log transformation of these times to generate a normal distribution. Then we did a linear mapping of these log transformations of times to get an exercise score ranging from 0 to 1. A 1 is given to students who spent the least time until getting

the answer correct and a 0 is given for students who could not solve the exercise correctly at all. Students who solved the exercise correctly in a time greater than a selected threshold for a certain exercise was given 0.2 in our case. For the programming exercises, the selected threshold time was more than double the time that we expected a typical student to use to solve the exercise in an exam.

- Tracing exercises case:
    1. The way that the data was collected from the exercises gives us no way to know which exercise from the group of exercises was actually attempted. We considered the time spent to get the first exercise correct as representative for the skill of the student on a given summary exercise. This argument is supported by the fact that tracing exercises from a given summary exercise requires the student to do the same task (e.g., trace a recursive function or detect infinite recursion, etc.) and the exercises within a given summary exercise are designed to be on the same difficulty level.
    2. We have noticed that students did not avoid certain tracing exercises, where they avoided certain writing exercises.
    3. We did a similar mapping on the writing exercises. We considered the time spent by the student until getting the question correct as a performance indicator. This is calculated as a function of both the time and the number of attempts done by the student. Then we have done a linear mapping from this calculated value to generate a score that ranges from 0 to 1. A 1 is given to the students who spent the least time until getting the answer correct and a 0 is given for the student who could not solve the exercise correctly at all. Students who solved the exercise correctly in a time greater than the selected threshold for a certain exercise was given 0.2. For all of the programming exercises, the selected threshold was more than double the time that we expected a typical student to use to solve the exercise in an exam.

Using the computed scores for writing and tracing exercises in RecurTutor, we have done an item analysis for all those exercises. Figures 5.3 and 5.4 show the item response curves (IRCs) for the writing and the tracing exercises respectively. For writing exercises, student ability was computed by the sum of the exercise scores for those writing exercises done by the student. We believe that the less the time spent to get the exercise correct the better the ability of the student and that's why we used exercise scores as the ability measure . For tracing exercises, student ability was computed by the summation of the score of each tracing done by the student in the tutorial because that's an indication of how well a student is in tracing skills of basic recursion. The IRCs shows that most of the exercises are predicting well student recursion skills. The exercises that were considered most hard did not predict well students recursion skills and were then taken off the the second version of the recursion tutorial. It was also shown from the log analysis for detecting the proficiency seeking behavior that the exercises that are shown to be difficult in the Item Response Curves are the same exercises that were avoided by the students who used proficiency-seeking gaming behavior. Appendix A shows the detailed difficulty and the discrimination indices for each of the exercises. In the next section, we consider correlations between exercise scores and exam scores.

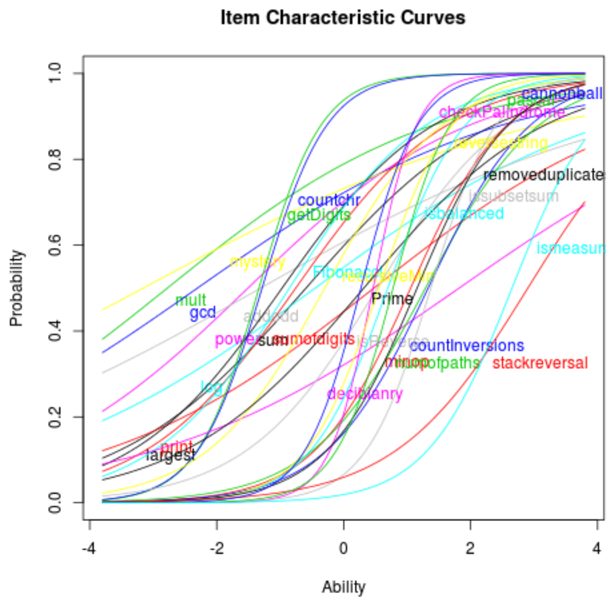


Figure 5.3: Item Response Curves for the writing programming exercises

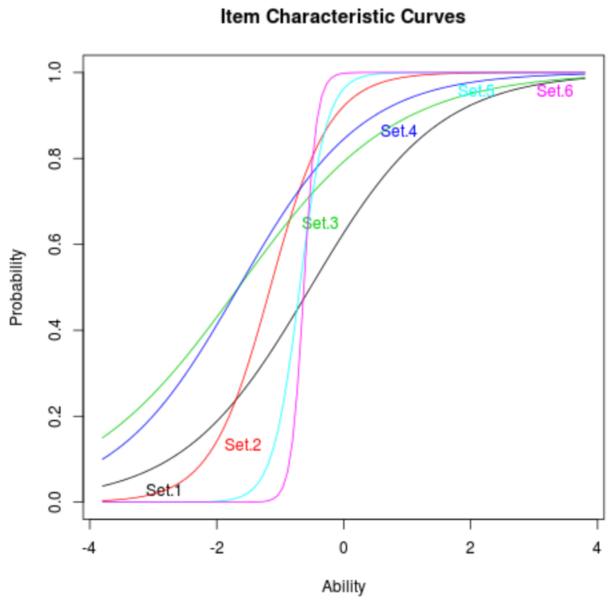


Figure 5.4: Item Response Curves for the tracing programming exercises

5.5.6 Correlation between performance on tutorial exercises and 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 (defined as 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 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 the semester exams. Quartile D represents students who performed below the 25^{th} percentile; Quartile C represents students with scores between the 25^{th} and the 50^{th} percentile; Quartile B represents students with scores between the 50^{th} and the 75^{th} percentile; and Quartile A represents students with scores above the 75^{th} 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 the exam scores over the semester. We are looking to determine if there is a relationship between writing and/or tracing performance and later success on writing and/or tracing questions within each quartile

We have done a Multivariate analysis of variance (MANOVA) 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 writing question?

Table 5.23: 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

Table 5.23 shows within each quartile how much does the number of writing or tracing exercises completed by student (as a performance measure) can predict their performance in the writing question in the final exam. Table 5.23 shows that, for all the quartiles, the number of writing exercises solved by the student significantly predict his performance on the writing question. It also shows that, for all quartiles, the number of tracing exercises significantly predict student performance on the writing question. Low performing students had 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 5.24: 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 5.24 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 5.24 shows that the number of tracing exercises significantly predict student performance on the tracing question in all quartiles except for the top quartile. Low performing student had more statistical significant correlation between the number of tracing exercises solved and the tracing question score.

We have repeated 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 overall exam score?

Table 5.24 showed that the number of writing exercises solved by the student did not predict 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 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

Table 5.25: 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

has students with the highest performance. That supports our driving hypothesis presented in 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, which gives an indication that enhancement of the scores of the writing and tracing questions in the final exam was actually caused by practicing more on the tutorial. This needs further investigation to see what are the factors in the exercises, specifically the tracing exercises, that support writing and tracing skills.

We performed a linear regression analysis to address items 3, 4 and 5 in this list. We want to see if the number of tracing and the 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 performed linear regression analysis to try to answer the following questions.

Does the number of writing exercises solved by a student predict student overall exam score, writing question score, or tracing question score?

Does the number of tracing exercises solved by a student predict student overall exam score, writing scores or tracing questions scores?

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, and the number of writing exercises solved and the writing score, and the number of writing exercises and the tracing question scores. All the p-values showed significantly low p-values, which means that changes in the predictor are associated to changes in the response variable. In our case, it emphasis 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 5.26 shows that the highest R-squared value (determination coefficient) was between

Table 5.26: 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 5.27: 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

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 results 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 writing score than that of the tracing exercises on the writing score but we believe that needs more study to investigate.

Table 5.26 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 inexplainable variability. For example, many psychology studies have R-squared values less that 50% because people are fairly unpredictable [28]. 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 5.27 shows the linear regression coefficients.

In summary, given our driving hypothesis presented in 3.4, we hypothesize that student performance in recursion, measured by the scores on recursion questions, can be enhanced by doing more practice. We have shown in this chapter 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 student can predict their scores on the final exam recursive writing and tracing questions but can not predict student performance on overall exam scores. That supports our hypothesis that the cause for better enhancement in student scores when using RecurTutor was due to doing more practice. We conclude that the best way to use RecurTutor to enhance student performance on recursion is to allow students to practice recursion on the tutorial. Further analysis is needed to understand what aspects of the practice exercises on RecurTutor leads to the enhancement in student performance.

5.6 Summary

In this chapter, we investigated the required versus the actual time spent on recursion by students who are using a typical instruction method. Then, we investigated the impact of using RecurTutor on the time spent, confidence level, and student scores. Specifically, we answered the following main research questions:

- **Does the amount of time that students spend in typical classes learning and practicing recursion match the amount of time that instructors believe to be required to learn and understand this topic?**
  From the survey results, students receiving traditional instructions do not spend enough time out-of-class practicing recursion and came out of the experience with a relatively low feeling of confidence. The instructors unanimously felt that students were not spending enough time.
- **Did using the basic recursion tutorial support basic recursion learning than the typical instruction?**
  From the t-test results, it was shown that the scores of the recursion questions for students who used RecurTutor were significantly greater than the students who have not used it.
- **Did using RecurTutor enhance the confidence level of the students on this topic?**
  From the t-test results, it was shown that the confidence level for the students who have used RecurTutor were significantly greater than the students who have not used it.
- **Did using our tutorial lead to students spending time in line with what instructors believe to be appropriate for learning and understanding basic recursion?**
  From the t-test results, it was shown the total time spent on recursion by students who used RecurTutor was not significantly more than the students who have not used it. However, the time spent on programming exercises was significantly more.

Then we have done an analysis of the log data for student use of RecurTutor. This analysis showed the following key results.

- Visualization skimming behavior, where students don't thoroughly read the slides of the visualizations, was common for most of the visualizations.
- Proficiency seeking behavior, where students game the system to skip hard exercises, was noticed in the hard writing programming exercises.

- No correlation was found between student performance, measured by the student score on the exams, and the proficiency seeking behavior nor the visualization skimming behavior.
- Students self-reported time on the use of RecurTutor is a reasonable match to the number computed from the actual use in the students logs.
- Students who did more writing exercises in RecurTutor were found to do better in the overall scores of the exams for the semester. The same was found for the tracing exercises.
- The more a student practice writing and tracing exercises the better is his score on the writing and tracing questions in the final exam. The number of the writing and tracing exercises solved by student was not correlated to overall exam scores.
- Better scores on the final exam writing recursion question was most correlated to solving more recursion tracing exercises.

Chapter 6

Basic Recursion Concept Inventory

This chapter reports on our initial attempts to develop a concept inventory that measures students misconceptions on basic recursion topics.

We present a collection of misconceptions and difficulties encountered by students when learning introductory recursion as presented in a typical CS2 course. A concept inventory in the form of a series of questions is provided in Appendix D, with the question rubric tagged to the list of misconceptions and difficulties.

A Concept Inventory is often associated with a set of misconceptions [83]. We have chosen instead to present a list of misconceptions and difficulties as discussed in Section 3.5.

The next sections show how we followed the traditional steps to build a draft basic recursion concept inventory.

6.1 Choose concepts

The first step in building a CI is to identify the concepts (topics) based on experts' rating for their difficulty and importance. Previous research has determined the most common problematic topics that lead to students' misunderstanding of recursion. For example, Sanders and Scholtz [90] claimed that a key factor in mastering recursion is understanding how the program moves from active control to the base case and then to the passive control in recursive functions. The complexity of the flow-of-control mechanism makes it a difficult concept for students to comprehend. It was found also that in most cases, students that have some difficulty with active flow are also confused about passive flow, and have misconceptions about the base case [93]. In addition, students are confused with the comparison to loop structures [4], and the lack of everyday analogies.

The following is a list of previously identified common problematic topics found in the literature for teaching recursion, ranked based on the frequency of appearance in the literature:

- Passive/backward control flow after reaching the base case [30, 90, 93] (Misconceptions: 1,2).
- The limiting case [30, 90] (Misconceptions: 7, 8 , 9, 10, 11).
- Active flow [30, 90].
- Comparison to loop structures [4].
- Variable updating either due to difficulty in evaluating a conditional statement or difficulty in understanding an explicit update statement [30] (Misconceptions:12).

We have started with the topics list that we have gathered from the literature then we have extended this list and broken some of the topics into multiple ones to be more descriptive and understandable. We have also changed the wording of some topics to be more clear. We have provided the new topics list with a description of each to 22 instructors to determine their opinions, then asked them to re-order with respect to how confusing they are to the students. We encouraged them to add, delete, merge, or re-word them if needed. The extended topics list presented to the instructors is as follows.

- Backward flow (BF): Passive control flow after reaching the base case.
- Active flow (AF): Active control flow until reaching the base case.
- Recursive calls (RC): How to formulate the recursive call.
- Limiting case (LC): How to formulate the stopping condition and when it will be triggered.
- Infinite recursion (INF): Wrongly write or call the recursive function so that the limiting case is never reached.
- Confusion with loop structure (LP): Implementing recursive functions (especially tail recursion) as a loop.
- Variables updating (VU): Unawareness of how variables are updated on every recursive call.

Two instructors were interviewed face to face, then we emailed 20 other instructors the list along with the instructions required. We received replies from 10 out of the 20. The instructors who replied were from five different institutions in three different countries. Appendix H shows the transcripts of the instructor interviews. Overall, 12 instructors agreed on the provided list and had minor modifications on the names or the order of the misconceptions.

The next step in building a CI is to identify the misconceptions. Section 3.5 discusses in details the process of identifying student misconceptions in basic recursion. The misconceptions found were then used to write the CI questions (items).

6.2 Write CI items and draft the CI

The third step in creating a CI is to write initial CI items (questions) based on the misconceptions generated from the previous step. We initially attempted to create a Concept Inventory as a series of multiple choice questions, with each question targeted to identify whether a student has a particular misconception or not. We quickly realized that a given multiple choice question with multiple distractors naturally relates to several misconceptions or difficulties, where each distractor ideally relates to a specific misconception or difficulty. We also realized that the nature of our topic lends itself to exercises where the student needs to determine the result of executing a piece of code.

It did not seem productive to limit the student to a specific list of distractors, as this would both "lead the witness" and also preclude discovering that students had previously unrecognized misconceptions or difficulties. Thus, all the questions are cast as "fill in the blank" (or free answer) questions, with a rubric that identifies the misconception or difficulty that would lead to a specific answer. If in the process of evaluating the answers to the concept inventory it is found that some answer not in the rubric is frequently given by students, this would suggest the need for further analysis to discover the cause. The initial rubrics for most of the questions are created from the answers that we have seen from approximately 8000 test responses. The first iteration of the draft CI questions and the misconceptions measured by each question can be found in the Appendix D.

6.3 Recursion CI Administration

The draft concept inventory was given to 23 students as a part of the mid-term exam in CS2114 during Summer II at Virgina-Tech. The mid-term exam had a total of 21 questions, of which 10 questions were on recursion (the concept inventory questions in Appendix D).

6.4 Reliability and Validity

6.4.1 CI Reliability

We measure the CI reliability using a single administration. The CI was given as a test to CS2114 students during Summer II. To measure single administration reliability we used Cronbach-α. The CS 2114 exam had a Cronbach-α reliability rating of 0.8. This preliminary finding indicates that the inventory has acceptable (above 0.5 is acceptable) single administration reliability after the first administration.

6.4.2 CI Validity

Student Content Validity

On our initial design of the rubric for each question, we tried to base the rubrics on patterns that we have seen in previous student responses to recursion questions. We checked the student answers on the test administered in Summer-II to see if, for each question, all candidate answers that we had in the rubrics have been given by the students. We have found that all the candidate answers in the questions rubrics were indeed given by the students. We found one answer that was not already covered by the rubric for each of question 1, 8 and 9, and so we have updated the rubrics to include those answers. We also found that Question 3 was answered correctly more than any other question. We think that the misconception covered by Question 3 is not widely held by the students. We looked at each of the 10 CI questions for all the 23 students, the student answer always exhibit one misconception. We believe that the design of the question and the rubric items make a reasonable grader agree that given a certain answers the student holds the matching misconception as listed in our rubrics. For each question, we have determined the corresponding misconceptions from our rubric for this question. If the answer is not found in the rubric, we add it to the rubric and tag it to one of the misconceptions that we have if possible. We counted the number of student answers that expresses each misconception. Table 6.1 shows the percentage of Summer II students who appear to hold each misconception.

Table 6.1: The percentage of students holding each Misconception based on the first CI administration

Misconception	**Percentage**
BFneverExecute	17.4%
BFexecuteBefore	13.04%
InfiniteExecution	4.35%
RCwrite	0%
RCnoReturnRequired	8.7%
RCreturnIsRequired	8.7%
BCbeforeRecursiveCase	8.7%
BCactionReturnConstant	4.35%
BCcheckAganistConstant	0%
BCwrite	0%
BCevaluation	21.75%
GlobalVariable	30.43%

The percentage of students holding each misconception is shown in Table 6.1, based on the scores of students who have used RecurTutor. A student was awarded the full score if the answer is correct, and zero if it exhibit any misconception. Using the tutorial should have some effect on their skill level, which would express itself as a change in those percentages. Thus, we do not expect the percentage of students holding each misconception to reflect the percentages from the pool of 8000 prior answers. In fact, that pool is mixed, including both pre-test and post-test responses (which should have different misconception percentages), with that post test coming from traditional instruction.

Table 6.2: Item Analysis

Question	**Difficulty Index**	**Discrimination Index**
item 1	56.52%	31.06%
item 2	59.48%	23.51%
item 3	95%	15.79%
item 4	91.3%	-4.8%
item 5	91.3%	47.19%
item 6	62.32%	42.05%
item 7	91.23%	58.28%
item 8	92.75%	54.39%
item 9	60.87%	22.67%
item 10	90.23%	63.6%

As part of validating our CI, we are interested in answering the following question:

Can one misconception hide another?

In other words, if a student solved a question with a certain answer such that we believe that this student holds the misconception corresponding to this answer on the rubric, can he also have other misconception that we can not detect because of the first one? Since not all of the misconceptions are covered by more than one item, if a student did not show a certain misconception on a certain question we need to be sure that this is not because another misconception is hiding the first. We have designed our rubrics so that each possible answer covered by the rubric is mapped to a misconception(s) that could not be hiding other misconceptions covered by the same question. A closer look at the rubrics of the concept inventory questions in Appendix D can show that. However, to answer this question accurately, the concept inventory can be expanded to have more questions so that a misconception is covered by more than one question. This will require a test that needs more time . That in turn makes administering the concept inventory harder because the instructors only allow limited access to students, which is bounded by the exam time.

We also checked to see if better performance on the entire CI correlated with better performance on individual questions, a process referred to as item response analysis [11]. We evaluated item quality by performing item analysis in Table 6.2 and constructing item re-

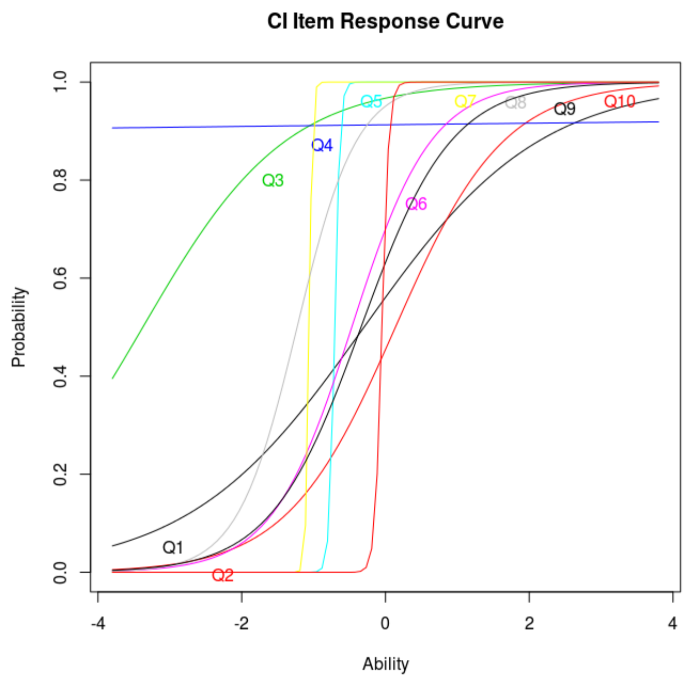


Figure 6.1: Item response curve for all the items in the recursion CI

sponse curves (IRCs) in Figure 6.1. The ability on the x-axis of the IRCs refers to student performance.

For most of the questions, the IRC demonstrates the desired correlation between conceptual knowledge and item performance. Student ability is measured by the sum of the scores of the ten concept inventory questions. For most of the questions, as the student ability increases, the probability to solve the question correctly increases as well. We found that question 4 did not show the desired behavior since student performance on the question did not vary according to student performance on the exam. This is why this question has a low discrimination index.

For future versions we decided to drop question 3, because in our investigation of previous recursion pre-tests, students can easily spot infinite recursion using their prior knowledge. Depending on expert feedback, we may also drop the items whose discrimination index is less than 30% (items 4 and 9). Since item 10 is a writing question, we plan to change it to make it a little bit harder (e.g., ask the students to implement a recursive function to find the largest element in an array, or to search for a given number in an array). The second iteration CI and the misconceptions measured by each question can be found in Appendix D. We will take the experts opinion on the new writing question before placing it in the new version of the concept inventory.

Expert Content Validity

We plan by the end of Fall 2015 to check content validity of the CI. We will collect feedback from experts. We will choose experts who have at least one year experience in teaching recursion. On individual items the experts will be asked to:

1. Answer the CI.
2. Decide whether the item reflects basic recursion concepts that students should know after completing a CS2 level course.
3. Rate the quality of the question.

Finally, the experts will be asked to provide their opinions about the CI as a whole. The experts will be asked to:

1. Decide if the CI as a whole reflects basic recursion knowledge after a CS2 level course
2. Comment on the topic coverage, and
3. Indicate how confident they would be that a student who performed well on the CI will perform well in basic recursion in a CS2 level course.

In order to accurately assess how reliable and valid the CI is, we need to administer it to a few thousand students at multiple institutions and to have feedback from experts from multiple institutions as well.

6.5 Summary

This chapter presents our initial efforts to define a collection of misconceptions and difficulties encountered by students when learning introductory recursion presented in a typical CS2 course. We have then presented first and second iterations of the draft concept inventory in the form of a series of questions, with the question rubric tagged to the list of misconceptions and difficulties. This initial effort should be continued by giving the CI to more students in different institutions and asking more experts to evaluate the CI. The reliability and the validity of the CI should be measured each time the CI is administrated to ensure that the developed recursion CI measures students' misconceptions on basic recursion. This recursion concept inventory is meant to measure student understanding of basic recursion regardless of the instruction method used. Our plan is to administer the CI in two institutions (Virginia Tech and Lehigh) in Fall 2015 and Spring 2015, and to gather experts feedback in Fall 2015.

Chapter 7

Advanced Recursion

This chapter presents our efforts to enhance the learning of recursion as it is typically taught post CS2. This is often cast in the form of recursive operations on binary trees. First, we have identified the misconceptions that students have in understanding recursion in binary trees through the analysis of exam responses and interviews. Second, we have designed a set of questions that should be usable as a pre-test or post-test to measure those misconceptions. We have designed a rubric for each question to match possible answers to misconceptions. Last, we have built a tutorial that addresses those misconceptions and trains students on avoiding them. The tutorial features code writing questions. It trains the students in part through a semantic code analyzer that detects misconceptions in the students' answers to the practice exercises, and provides appropriate feedback.

7.1 Identifying misconceptions

The only work related to identifying student misconceptions in the context of binary trees (by Karpierz et. al [52]) discussed in Section 2.4.2. It was not related to recursion.

Murphy et. al's work discussed in Section 2.2.5 analyzed student answers for a problem to write a recursive method to traverse a Binary Search Tree and count the number of nodes that has exactly one child. However, Murphy et. al's work is not related to specific misconceptions in binary trees. Instead, it is related to general difficulties in writing a recursive method like writing a correct base case or recursive call.

Since we have not found any work in the literature that explicitly discusses student misconceptions in the context of recursion in binary trees, we have conducted student interviews and analyzed student answers to recursion in binary trees questions to come up with a list of the common misconceptions for this topic.

7.1.1 Student interviews

From the Fall 2014 offering of a CS3 course, we have reviewed the midterm exam answers, for the question that targets recursion in binary tees. This question asks the student to write a recursive function that, given the root to a Binary Search Tree and a key, returns the number of nodes having key values less than K. We selected a pool of students who have answered the question incorrectly or inefficiently and sent the students an interview invitation email (Appendix G). Four students agreed to participate. We tried to determine why they solved the mid-term question incorrectly or inefficiently, and how can we help them to avoid these misconceptions. The interview questions and the transcript from the interviews can be found in Appendix I.2. We conclude from the interviews that the main reason for the misconceptions is lack of practice with appropriate feedback that consistently warns the student about their misconceptions. For the students to overcome the misconceptions, we believe that they must solve more programming exercises on recursion in binary trees, with better feedback on their misconceptions. That was our inspiration for building the binary tree recursion tutorial with semantic code analysis to provide this kind of feedback. We call this resulting tutorial BTRecurTutor.

7.1.2 Student exam response analysis

We analyzed student answers to test questions and student responses to an automatically assessed programming exercise on recursion in binary trees. We have analyzed more than 600 responses for binary tree recursive function writing questions given to students over three semesters in pre-test, post-test, mid-term, or final exams of a CS3 level Data Structures and Algorithms course. In addition, we have analyzed more than 5200 attempts from 640 students in 3 institutions over 3 semesters on an automatically assessed programming exercise on recursion in binary trees. The programming exercise asks the students to write a recursive function to count the number of leaf nodes in a given binary tree. From that, we found common misconceptions and difficulties encountered by students when writing functions that involves using recursion to traverse a binary tree.

We can identify differences between the misconceptions related to recursion in binary trees and the misconceptions that we identified for basic recursion. Most of the misconceptions related to binary trees do not necessarily lead to wrong outputs (i.e., the resulting function will pass the unit tests). Instead, many of these misconceptions tend to relate to any code complexity or algorithmic inefficiency.

The following is the list of the common misconceptions and difficulties that we have found.

1. In any recursive function that traverses a binary tree, we must explicitly check if the children of the current node is null or not before making the recursive call [childIsNull]
2. In any recursive function that traverses a binary tree, if we check the current node's value then we have to explicitly check its children's values. [childCheckValue]

3. In any recursive function that traverses a binary tree, we have to explicitly check whether the current node is a leaf or not to terminate the recursive function. [rootIsLeaf]
4. In any recursive function that traverses a binary tree, we do not need to check if the root is Null. [rootIsNotNull]
5. [Difficulty] In a recursive function that traverses a BST, we have to traverse the whole tree regardless of the aim of the traversal. For example, when searching for the minimum value in the tree, student check the right subtree. [BSTMinCheckRight]
6. [Difficulty] In a recursive function that traverses a BST, we may not traverse the whole tree. For example, when searching for the minimum value in the tree, student (sometimes or always) fail to search the left subtree. [BSTMinNoCheckLeft]

Note that only misconception 4 actually results in a function that gives a wrong answer (and then only if the initial call is made with a null tree). However, it is important for students to learn to avoid unnecessary code complexity. Otherwise, they will find it more difficult or impossible to write more complicated recursive functions, such as operations on advanced data structures like 2-3 trees.

7.2 Questions to test student understanding of recursion in binary trees

Appendix E shows the questions that we have used to evaluate student understanding of recursion in binary tree. Each question rubric is tagged to the list of misconceptions and difficulties. We assume that the student solving the test should already be proficient in basic recursion, so we limit our rubrics to the misconceptions related to recursion in binary trees. Through analyzing student answers to recursion in binary tree questions, we found that students do in fact express misconceptions related to recursion in binary trees more than those related to basic recursion. Roughly 30% of the wrong answers were due to problems related to understanding basic recursion, while the other 70% were due to misconceptions related to recursion in binary trees.

7.3 Advanced Recursion in Binary Trees Tutorial

Finding student misconceptions and difficulties in advanced recursion in binary trees inspired us to build an online tutorial on this topic. The tutorial is a significant expansion of material originally appearing in [95] but which did not in its original form properly address typical student misconceptions. The tutorial's textual content and visualizations try to explicitly explain to students the common misconceptions and how to avoid them. The tutorial features practice programming exercise that help students to practice more on recursion in binary trees. In the programming exercises infrastructure, we implemented semantic code analysis that detects a student's misconceptions from his answer, and provides appropriate detailed feedback that warns the student about the misconception encountered.

7.3.1 Tutorial Content

The tutorial content was reviewed and refined by 4 instructors. Each of the instructors has more than ten years of experience in teaching advanced recursion in binary trees. Figure 7.1 shows an example of a lesson in the tutorial.

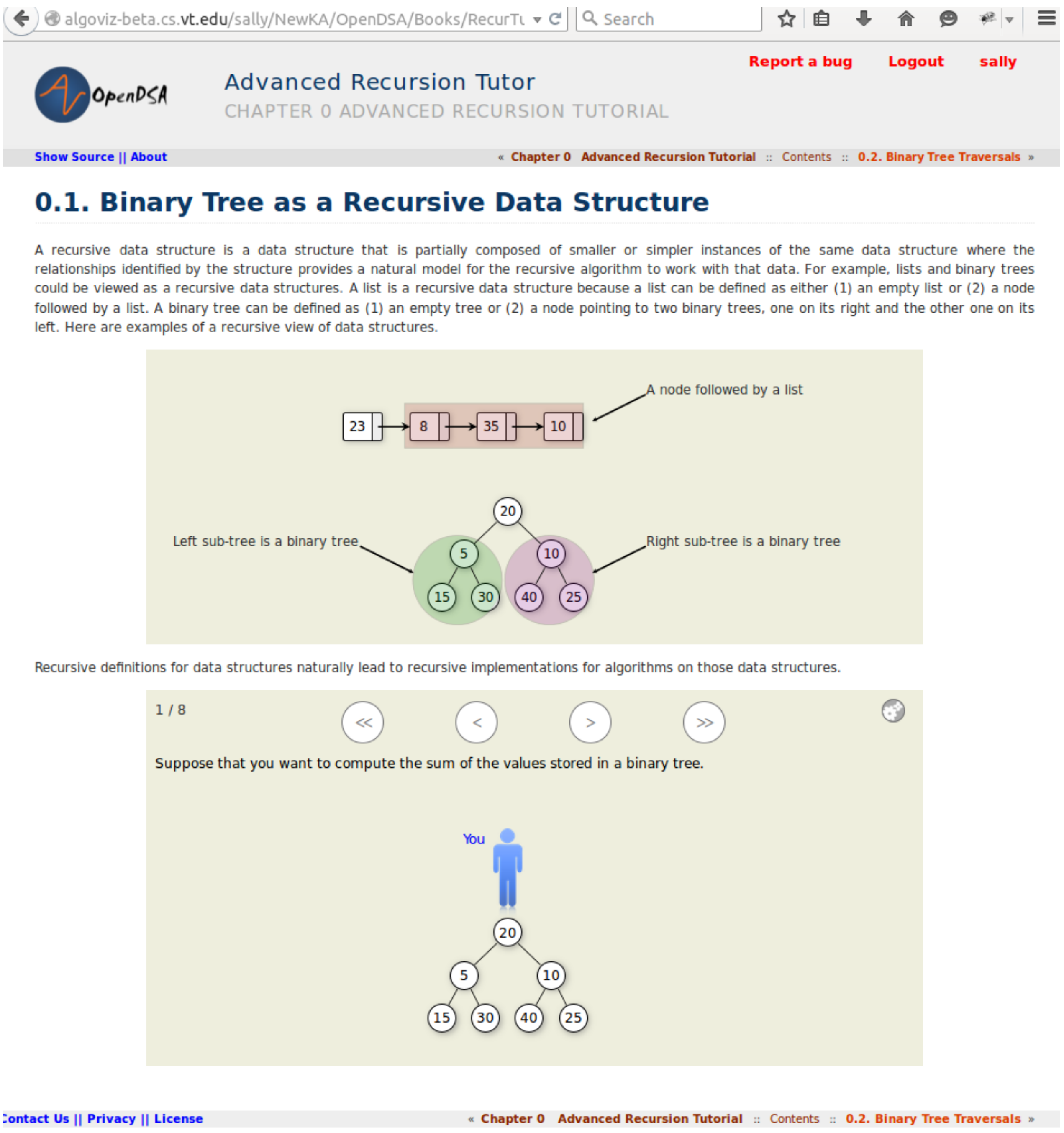
The tutorial is divided into the following modules.

1. Binary Tree as a Recursive Data Structure: Shows how we view a binary tree as a recursive data structure, and how that leads to recursive implementation for the operations done on the binary tree.
2. Binary Tree Traversals: Shows different ways to enumerate all binary tree nodes. It covers preorder, inorder and, postorder traversals.
3. Implementing Tree Traversals: Shows the detailed steps needed to write a recursive function that traverses a binary tree. It covers how to write a base case , its action, a recursive call and its action in the context of binary trees traversals.
4. Information Flow in Recursive Functions: Illustrates how to handle the different types of information flow in a recursive function that traverses a binary tree. The module shows the different types of information flow: local, passing down information, collect and return, and combined information flows.
5. Binary Search Trees: Introduces the Binary Search Tree. It also shows (using visualizations) how to search, insert, and remove a value in a BST.
6. Binary Tree Guided Information Flow: Covers guided information flow, which is most relevant to operations on BSTs.
7. Multiple Binary Trees: Practice exercises that ask the student to implement recursive functions that perform operations on two binary trees.
8. Hard Information Flow Problems: Shows an example of how to test if a given tree is a BST. In this example, the solution is not based on local information but it depends on passing relevant information down the tree.

Visualizations

In order to provide explicit instruction to combat typical misconceptions, we added the following visualizations to OpenDSA's binary trees chapter.

1. Sum on a binary tree: Focuses on the abstraction of recursion. This visualization uses the delegation process discussed in [19]. It shows an example of how to compute the sum of the values of all the nodes in a binary tree by delegating the task to two friends.



algoviz-beta.cs.vt.edu/sally/NewKA/OpenDSA/Books/RecurTu

Report a bug Logout sally

Advanced Recursion Tutor

CHAPTER 0 ADVANCED RECURSION TUTORIAL

Show Source || About « Chapter 0 Advanced Recursion Tutorial :: Contents :: 0.2. Binary Tree Traversals »

0.1. Binary Tree as a Recursive Data Structure

A recursive data structure is a data structure that is partially composed of smaller or simpler instances of the same data structure where the relationships identified by the structure provides a natural model for the recursive algorithm to work with that data. For example, lists and binary trees could be viewed as a recursive data structures. A list is a recursive data structure because a list can be defined as either (1) an empty list or (2) a node followed by a list. A binary tree can be defined as (1) an empty tree or (2) a node pointing to two binary trees, one on its right and the other one on its left. Here are examples of a recursive view of data structures.

A node followed by a list
23
8
35
10
Left sub-tree is a binary tree
Right sub-tree is a binary tree
20
5
10
15
30
40
25

Recursive definitions for data structures naturally lead to recursive implementations for algorithms on those data structures.

1 / 8

Suppose that you want to compute the sum of the values stored in a binary tree.

Contact Us || Privacy || License « Chapter 0 Advanced Recursion Tutorial :: Contents :: 0.2. Binary Tree Traversals »

Figure 7.1: Example of a lesson in BTRecurTutor.

2. Common Mistakes: Shows, using code examples, the common misconceptions that students encounter when writing a recursive function that traverses a binary tree.
3. BST Common Mistakes: Traces an example of a recursive function on a BST to show the common mistakes that students display. It shows how to benefit from the BST properties when writing a recursive function, so as to traverse fewer nodes.

Programming Exercises

All programming exercises in this tutorial ask student to write a full function that performs a certain task. The programming exercises train the student on different types of information flows, or how to deal with multiple binary trees. The programming exercises fall into the following categories:

1. Local: This type of traversal involves going to each node in the tree to do some local operation. Such tasks need no information flow between the binary tree nodes.
2. Passing Down Information: This type of traversal, involves passing the same information to every node in the binary tree.
3. Collect and return: This type of traversal, requires that we communicate information back up the tree to the caller.
4. Combining Information Flows: This type of traversal requires both that information be passed down, and that information be passed back.
5. Guided: This type of traversal should not visit every node in the tree. This means that the recursive function is making some decision at each node that sometimes allows it to avoid visiting one or both of its children. The decision is typically based on the value of the current node. Many problems that require information flow on BSTs are considered to be guided.
6. Multiple Binary Trees: This type of problem involves operations on more than one binary tree.

Appendix B shows a detailed description of the examples that are provided in BTRecurTutor.

7.4 Semantic Code Analysis

BTRecurTutor uses the same programming exercise infrastructure from RecurTutor which is described in Section 4.3. We use the same feedback categories as used by the writing exercises of RecurTutor (shown in Subsection 4.2.3). For BTRecurTutor, we added to the programming exercise infrastructure a semantic code analyzer that detects misconceptions related to recursion in binary trees in the student answer and provide detailed feedback on the misconception that the student displayed in his answer.

The testing main process first compiles the code. If the code compiles successfully, then the infrastructure checks if the output is correct or not. If the code is correct, then the semantic code analyzer is called, which is implemented as a python function. The semantic analyzer is passed the student's code and the exercise name. The semantic code analyzer checks an ad hoc set of misconceptions, depending on the given problem. If the semantic code analysis finds evidence of a misconception, then the answer is considered incorrect and feedback from the semantic code analysis is returned back to the testing main process to the be sent to the client be shown to the student.

For example, if the exercise is asking to find if a certain value exists in a given tree, the following will be checked by the semantic code analysis:

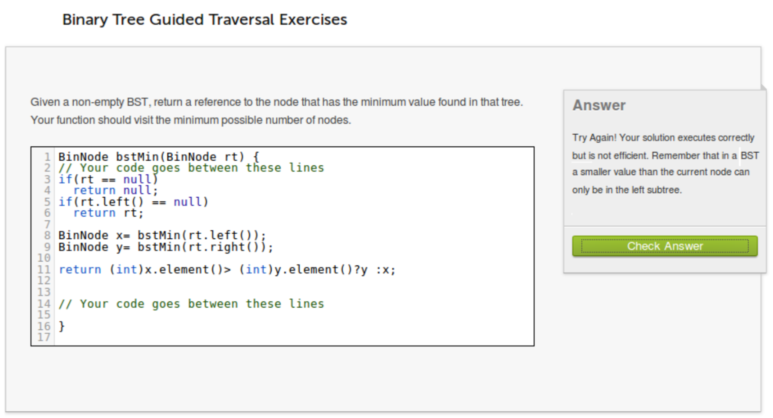
1. Does the student check if the root is null? This required to successfully pass all tests cases.
2. Does the student explicitly check on the child(ren) value(s)? Such checks are unnecessary for this problem.
3. Does the student explicitly check if the child(ren) are null? Such a check is redundant for this problem due to (1).

In the post-testing phase, feedback is sent back to the client, which is then displayed to the student. Currently, semantic code analysis is implemented for about 15 binary tree programming exercises.

Depending on the exercise, the semantic code analyzer checks on certain misconceptions in the student's code. The semantic code analyzer can detect any of the following misconceptions:

1. Unnecessarily check if the children of the current node are null or not.
2. Unnecessarily check the children values whenever checking the current node's value.
3. Unnecessarily check whether the current node is a leaf or not to terminate the recursive function.
4. Missing a check to see if the root is null.
5. In a recursive function that traverses a BST, process sub-trees that cannot contain nodes with the target property. For example, when searching for the minimum value in the tree, the function should not check the right subtree.
6. In a recursive function that traverses a BST, miss processing sub-trees that contain nodes with the target property. For example, when searching for the minimum value in the tree, sometimes (or always) fail to search the left subtree.

Detailed feedback is provided to the student based on the misconception(s) displayed in his response to the programming question. Figure 7.2 shows an example of the detailed feedback provided from the semantic code analysis.



Binary Tree Guided Traversal Exercises

Given a non-empty BST, return a reference to the node that has the minimum value found in that tree. Your function should visit the minimum possible number of nodes.

```
BinNode bstMin(BinNode rt) {
// Your code goes between these lines
if(rt == null)
  return null;
if(rt.left() == null)
  return rt;

BinNode x= bstMin(rt.left());
BinNode y= bstMin(rt.right());

return (int)x.element()> (int)y.element()?y :x;


// Your code goes between these lines

}

```

Answer

Try Again! Your solution executes correctly but is not efficient. Remember that in a BST a smaller value than the current node can only be in the left subtree.

Figure 7.2: Example of feedback from semantic code analysis.

7.5 Evaluation Plan

In Fall 2015, CS3114 Data Structures and Algorithms students are using a version of the advanced tutorial that has two exercises that warn the students about their misconceptions. In Spring 2016, CS3114 students will be given the advanced tutorial that has about 12 exercises that warn the students about their misconceptions. We will then compare the results from those two semesters to the baseline data we had from Fall 2014 and Spring 2015. Our goal is to answer the following question:

Does warning the students about certain misconceptions for the topic of recursion on binary trees lessen the frequency of occurrence of those misconceptions in student answers in the exam questions related to this topic?

7.6 Summary

This chapter presents our efforts to enhance the learning of recursion in binary trees. First, we have identified the misconceptions that students have in understanding recursion in binary trees based on student interviews and a review of their answers to midterm and final exam questions. Second, we have designed a set of questions that could act as a pre-test and post-test to measure those misconceptions. We have designed a rubric for each question to match possible answers to misconceptions. Last, we have built a tutorial that addresses those misconceptions. The tutorial also trains students to avoid those misconceptions through a semantic code analyzer that detects the misconceptions in the students' answers to the practice exercises in the tutorial, and gives appropriate feedback.

Chapter 8

Conclusion and Future Work

In this chapter we summarize the dissertation by presenting the problems that we addressed, and our contributions. We conclude this chapter by describing future research directions.

The main goal of this work is to enhance the learning of recursion. Recursion is one the most important and hardest topics in lower division computer science courses. However, none of the work in the literature addressed the effect of doing more practice on learning recursion, nor did there previously exist a concept inventory to measure student learning of recursion.

On one side, we built two recursion tutorials to enhance student learning of this topic through addressing the main misconceptions, and through allowing students to do more automatically graded practice than was practical before. On the other side, we built a recursion draft concept inventory to independently assess student understanding of recursion, regardless of the method of instruction used to teach it.

8.1 Contributions

Next we briefly describe each of our contributions.

1. A study that showed a gap between the time that an instructor expected a student to practice recursion out-of-class and what time students actually spend, when given traditional instruction.
2. A new teaching approach for recursion based on greater student interaction with the material than has previously been possible, presented in the form of two OpenDSA tutorials.
3. A study to determine the effect of more practice on learning recursion.
4. Exercises that address the main common misconceptions that students have in learning recursion.
5. Infrastructure to support automatic assessment for programming exercises, which will

help by giving the students immediate feedback without putting additional grading burden on the instructor.

6. A draft concept inventory for measuring student understanding of basic recursion. A plan is in place for evaluating and refining the draft CI.
7. An analysis to find out the most common misconceptions in understanding recursion in binary trees.
8. Infrastructure to support semantic code analysis of students answers for programming exercises, which gives students a detailed feedback on their misconceptions.
9. A complete tutorial to help student learn basic recursion.
10. A complete tutorial to help student learn recursion in binary trees. A plan is in place for evaluating the advanced recursion tutorial.

8.2 Future work

In Chapters 6 and 7, we indicated specific activities that are already planned and will be executed to validate the concept inventory and the two tutorials. In this section, we identify additional avenues for future research.

1. We need further investigation on which of the factors or combination of factors caused the enhancement to student's scores and confidence level on basic recursion when RecurTutor was used. That could be done by collecting more data on student use of CodingBat in other control groups.
2. We have seen that students who did more practice on the tracing exercises got better grades on the writing question in the exam. More investigation is needed to know the reason for the enhancement.
3. We have seen that students who did more practice on the writing exercises got better grades on the exams. More investigation is needed to know the reason for the enhancement.
4. 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 study the effect of enhancing the tracing ability of the student on the writing ability and vice versa. As a future experiment, we can have a group that solves only tracing exercises and another group that solves only writing exercises. Then the two groups should solve the same writing and tracing questions. We can then compare the performance of the two groups.
5. The basic recursion concept inventory needs further refinement. It needs to be administered at different institutions and to thousands of students. Expert feedback is

required at every iteration. Concept inventory reliability and validity need to be measured to make sure that we have a valid and a reliable CI that can measure student understanding of basic recursion irrespective of the used instruction.

6. We have built a tutorial that addresses the main misconceptions in recursion in binary trees and detects the student misconceptions in submitted answers. For this tutorial, further experimentation is needed to know if warning the students about certain misconceptions on the topic of recursion in binary trees lessens the frequency of the occurrence of those misconceptions in student answers in the exam questions related to this topic. We need also to know whether doing more practice that warns the student about their misconceptions changes the frequency of occurrence for those misconceptions in student answers in the exam questions related to this topic.
7. We have built a test to measure student understanding of advanced recursion in binary trees. For each question in that test, we designed rubrics such that each possible answer is tagged to a misconception. This test needs further refinement to become a concept inventory that measures student understanding of this topic.

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 questions to measure student understanding of hard programming skills.

Bibliography

[1] M. J. Allen and W. M. Yen. *Introduction to measurement theory.* Waveland Press, 2001.

[2] V. L. Almstrum, P. B. Henderson, V. Harvey, C. Heeren, W. Marion, C. Riedesel, L. Soh, and A. Tew. Concept inventories in computer science for the topic discrete mathematics. *SIGCSE Bulletin*, 38(4):132–145, June 2006.

[3] J. Andrus and J. Nieh. Teaching operating systems using Android. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 613–618, 2012.

[4] A. C. Benander and B. A. Benander. Student monks–teaching recursion in an IS or CS programming course using the towers of hanoi. *Journal of Information Systems Education*, 19(4):455–467, 2008.

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

[6] M. Blumenstein, S. Green, A. Nguyen, and V. Muthukkumarasamy. An experimental analysis of GAME: a generic automated marking environment. *SIGCSE Bulletin*, 36(3):67–71, June 2004.

[7] J. R. Buck, K. E. Wage, M. A. Hjalmarson, and J. K. Nelson. Comparing student understanding of signals and systems using a concept inventory, a traditional exam and interviews. In *Frontiers In Education Conference-Global Engineering: Knowledge Without Borders, Opportunities Without Passports, 2007. FIE'07. 37th Annual*, pages S1G–1, 2007.

[8] A. Chaffin, K. Doran, D. Hicks, and T. Barnes. Experimental evaluation of teaching recursion in a video game. In *Proceedings of the 2009 ACM SIGGRAPH Symposium on Video Games*, Sandbox '09, pages 79–86, 2009.

[9] TH. Chi, R. Glaser, and M. Farr. *The nature of expertise.* Psychology Press, 2014.

[10] R. C. Clark and R. E. Mayer. *E-learning and the science of instruction: Proven guidelines for consumers and designers of multimedia learning.* Wiley. com, 2011.

[11] L. Crocker and J. Algina. *Introduction to classical and modern test theory.* ERIC, 1986.

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

[13] N. B. Dale and C. Weems. *Introduction to Pascal and structured design.* 1987.

[14] N. Dalkey and O. Helmer. An experimental application of the Delphi method to the use of experts. *Management science*, 9(3):458–467, 1963.

[15] H. Danielsiek, W. Paul, and J. Vahrenhold. Detecting and understanding students' misconceptions related to algorithms and data structures. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 21–26. ACM, 2012.

[16] W. Dann, S. Cooper, and R. Pausch. Using visualization to teach novices recursion. *SIGCSE Bulletin*, 33(3):109–112, June 2001.

[17] C. D'Avanzo. Biology concept inventories: overview, status, and next steps. *Bio-Science*, 58(11):1079–1085, 2008.

[18] J. Dong, Y. Sun, and Y. Zhao. Design pattern detection by template matching. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 765–769, 2008.

[19] J. Edgington. Teaching and viewing recursion as delegation. *Journal of Computing Sciences in Colleges*, 23(1):241–246, October 2007.

[20] J. Eldred, J. Ward, K. Snowden, and Y. Dutton. The nature and role of confidence-ways of developing and recording changes in the learning context. *Adults Learning Journal*, 2006.

[21] J. Eskola and J. Tarhio. On visualization of recursion with Excel. In *Proceedings of the Second Program Visualization Workshop*, pages 45–51, HorstrupCentret, Denmark, June 2002.

[22] G. Ford. A framework for teaching recursion. *SIGCSE Bulletin*, 14(2):32–39, June 1982.

[23] G. Ford. An implementation-independent approach to teaching recursion. *SIGCSE Bulletin*, 16(1):213–216, January 1984.

[24] E. Fouh. *Building and Evaluating a Learning Environment for Algorithm and Data Structures Courses.* PhD thesis, Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 2015.

[25] E. Fouh, V. Karavirta, D. Breakiron, S. Hamouda, S. Hall, T. Naps, and C. A. Shaffer. Design and architecture of an interactive etextbook–The OpenDSA system. *Science of Computer Programming*, 88:22–40, 2014.

[26] D. P. Friedman. *The Little Schemer*. The MIT Press, 1996.

[27] D. P. Friedman, M. Felleisen, and G. L Steele. *The Little LISPer*. MIT Press Cambridge, MA, 1987.

[28] Jim Frost. How to interpret a regression model with low r-squared and low p values, 2014.

[29] T. S. Gegg-Harrison. Exploiting program schemata to teach recursive programming. *P. Brna, B. duBoulay, and H. Pain (eds.), Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study*, pages 347–379, 1999.

[30] C. E. George. EROSI-visualising recursion and discovering new errors. *SIGCSE Bulletin*, 32(1):305–309, March 2000.

[31] A. Gerdes, J. T. Jeuring, and B. J. Heeren. Using strategies for assessment of programming exercises. *SIGCSE Bulletin*, 40(4):441–445, November 2010.

[32] D. Ginat. Do senior CS students capitalize on recursion? In *SIGCSE Bulletin*, volume 36, pages 82–86. ACM, 2004.

[33] D. Ginat and E. Shifroni. Teaching recursion in a procedural environmenthow much should we emphasize the computing model? *SIGCSE Bulletin*, 31(1):127–131, 1999.

[34] Y. S. Give'on. Is recursion well defined?? *Computer Education*, 14(1):35–41, January 1990.

[35] K. Goldman, P. Gross, C. Heeren, G. Herman, L. Kaczmarczyk, M. C. Loui, and C. Zilles. Identifying important and difficult concepts in introductory computing courses using a Delphi process. *SIGCSE Bulletin*, 40(1):256–260, 2008.

[36] K. Goldman, P. Gross, C. Heeren, G. L. Herman, L. Kaczmarczyk, M. C. Loui, and C. Zilles. Setting the scope of concept inventories for introductory computing subjects. *Transactions of Computing Education*, 10(2):5:1–5:29, June 2010.

[37] A. Gordon. Teaching recursion using recursively-generated geometric designs. *Journal of Computing Sciences in Colleges*, 22(1):124–130, October 2006.

[38] T. Götschi, I. Sanders, and V. Galpin. Mental models of recursion. In *Proceedings of the fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, pages 103–107, 2010.

[39] G. L. Gray, D. Evans, P. Cornwell, F. Costanzo, and B. Self. Toward a nationwide dynamics concept inventory assessment test. In *American Society for Engineering Education Annual Conference & Exposition*, 2003.

[40] J. E. Greer. *An empirical comparison of techniques for teaching recursion in introductory computer sciences*. PhD thesis, University of Texas at Austin, 1987.

[41] K. Gunion, T. Milford, and U. Stege. Curing recursion aversion. *SIGCSE Bulletin*, 41(3):124–128, July 2009.

[42] P. J. Guo. Online Python tutor: Embeddable web-based program visualization for CS education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM.

[43] M. T. Harandi and J. Q. Ning. Knowledge-based program analysis. *IEEE Software*, 7(1):74–81, January 1990.

[44] M. T. Helmick. Interface-based programming assignments and automatic grading of java programs. In *Proceedings of the twelfth annual conference on Innovation and technology in computer science education*, ITiCSE '07, pages 63–67, 2007.

[45] G. L Herman. *The development of a digital logic concept inventory*. PhD thesis, University of Illinois at Urbana-Champaign, 2011.

[46] G. L. Herman, M. C. Loui, and C. Zilles. Creating the digital logic concept inventory. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 102–106. ACM, 2010.

[47] W. Hsin. Teaching recursion using recursion graphs. *Journal of Computer Sciences Colleges*, 23(4):217–222, 4 2008.

[48] P. N. Johnson-Laird. *Mental models: towards a cognitive science of language, inference, and consciousness*. 1983.

[49] L. C. Kaczmarczyk, E. R. Petrick, J. P. East, and G. L Herman. Identifying student misconceptions of programming. *SIGCSE Bulletin*, pages 107–111, 2010.

[50] H. Kahney. What do novice programmers know about recursion. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '83, pages 235–239, 1983.

[51] V. Karavirta and P. Ihantola. Serverless automatic assessment of Javascript exercises. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*, ITiCSE '10, page 303, 2010.

[52] K. Karpierz and S. A. Wolfman. Misconceptions and concept inventory questions for binary search trees and hash tables. In *Proceedings of the 45th ACM technical symposium on Computer science education*, SIGCSE '14, pages 109–114. ACM, 2014.

[53] C. M. Kessler and J. R. Anderson. Learning flow of control: recursive and iterative procedures. *Human Computer Interaction*, 2(2):135–166, June 1986.

[54] `http://khanacademy.org`, 2012.

[55] E. B. Koffman. *Pascal(4th Edition)*. 1992.

[56] A. Korhonen, L. Malmi, P. Silvasti, J. Nikander, P. Tenhunen, P. Mard, H. Salonen, and V. Karavirta. TRAKLA2. `http://www.cs.hut.fi/Research/TRAKLA2/`, 2003.

[57] S. Krause, J. Birk, Ri. Bauer, B. Jenkins, and M. J. Pavelich. Development, testing, and application of a chemistry concept inventory. In *Frontiers in Education, 2004. FIE 2004. 34th Annual*, pages T1G–1. IEEE, 2004.

[58] J. Krone, J. E. Hollingsworth, M. Sitaraman, and J. O. Hallstrom. A reasoning concept inventory for computer science. *Clemson University*, 2010.

[59] R. L. Kruse. On teaching recursion. *SIGCSE Bulletin*, 14(1):92–96, February 1982.

[60] C. M. Lewis. Exploring variation in students' correct traces of linear recursion. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, ICER '14, pages 67–74, 2014.

[61] L. Llana, E.e Martin-Martin, C. Pareja-Flores, and J. Velázquez-Iturbide. FLOP: A user-friendly system for automated program assessment. *Journal of Universal Computer Science*, 20(9):1304–1326, 2014.

[62] P. Longo, A. Sterbini, and M. Temperini. *TSW: A Web-Based Automatic Correction System for C Programming Exercises*. Springer, 2009.

[63] L. Malmi, V. Karavirta, A. Korhonen, J. Nikander, O. Seppälä, and P. Silvasti. Visual algorithm simulation exercise system with automatic assessment: TRAKLA2. *Informatics in Education*, 3(2):267–288, September 2004.

[64] S. Maniccam. Towers of Hanoi related problems. *Computing Sciences in Colleges*, 27(5):205–213, May 2012.

[65] R. E. Mayer. The psychology of how novices learn computer programming. *ACM Computing Surveys*, 13(1):121–141, March 1981.

[66] R. E. Mayer and P. Bayman. Using conceptual models to teach basic computer programming. *Journal of Educational Psychology*, 80(3):291–298, September 1988.

[67] R. McCauley, B. Hanks, S. Fitzgerald, and L. Murphy. Recursion vs. iteration: An empirical study of comprehension revisited. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, pages 350–355, 2015.

[68] J. Mikko, S. Tapio, and K. Ari. The feasibility of automatic assessment and feedback. In *International Conference on Cognition and Exploratory Learning in Digital Age*, pages 113–122, 2005.

[69] B.N. Miller and D.L. Ranum. Beyond PDF and ePub: toward an interactive textbook. In *Proceedings of the seventeenth ACM Annual Conference on Innovation and Technology in Computer Science Education (ITiSE'12)*, pages 150–155, 2012.

[70] C. Mirolo. Learning (through) recursion: a multidimensional analysis of the competences achieved by CS1 students. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*, ITiCSE '10, pages 160–164, 2010.

[71] L. Murphy, S. Fitzgerald, S. Grissom, and R. McCauley. Bug Infestation!: A goal-plan analysis of CS2 students' recursive binary tree solutions. *SIGCSE Bulletin*, pages 482–487, 2015.

[72] M. A. Nelson, M. R. Geist, R. L. Miller, R. A. Streveler, and B. M. Olds. How to create a concept inventory: The thermal and transport concept inventory. In *Annual Conference of the American Educational Research Association, Chicago, IL*, 2007.

[73] M. L. Nelson. A survey of reverse engineering and program comprehension. *arXiv preprint cs/0503068*, 2005.

[74] M. Norman and T. Hyland. The role of confidence in lifelong learning. *Educational studies*, 29(2-3):261–272, 2003.

[75] N. Parlante. Codingbat: code practice. 2011.

[76] W. Paul and J. Vahrenhold. Hunting high and low: Instruments to detect misconceptions related to algorithms and data structures. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 29–34, 2013.

[77] P. L. Pirolli and J. R. Anderson. The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology/Revue canadienne de psychologie*, 39(2):240–272, 1985.

[78] I. Polycarpou, A. Pasztor, and M. Adjouadi. A conceptual approach to teaching induction for computer science. *SIGCSE Bulletin*, 40(1):9–13, 3 2008.

[79] D. Pritchard and T. Vasiga. CS circles: An in-browser Python course for beginners. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 591–596, 2013.

[80] N. Ragonis and M. Ben-Ari. *A long-term investigation of the comprehension of OOP concepts by novices.* Taylor & Francis, 2005.

[81] C. Rinderknecht. A survey on teaching and learning recursive programming. *Informatics in Education*, 13(1):87–119, 2014.

[82] E. Roberts. *Thinking recursively.* J. Wiley, 1986.

[83] G. Rowe and C. Smaill. Development of an electromagnetic course-concept inventory-a work in progress. In *Proceedings of the eighteenth Conference of Australian Association for Engineering*, 2007.

[84] M. Rubio-Sánchez. Tail recursive programming by applying generalization. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*, ITiCSE '10, pages 98–102, 2010.

[85] M. Rubio-Sánchez and I. Hernán-Losada. Exploring recursion with fibonacci numbers. *SIGCSE Bulletin*, 39(3):359–359, June 2007.

[86] M. Rubio-Sánchez, J. Urquiza-Fuentes, and C. Pareja-Flores. A gentle introduction to mutual recursion. *SIGCSE Bulletin*, 40(3):235–239, June 2008.

[87] M. Rubio-Sánchez and J. Velázquez-Iturbide. Tail recursion by using function generalization. In *Proceedings of the fourteenth annual ACM SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '09, pages 394–394, 2009.

[88] S. Rugaber. Program comprehension. *Encyclopedia of Computer Science and Technology*, 35(20):341–368, 1995.

[89] R. Saikkonen, L. Malmi, and A. Korhonen. Fully automatic assessment of programming exercises. In *Proceedings of the sixth annual conference on Innovation and technology in computer science education*, ITiCSE '01, pages 133–136, 2001.

[90] I. Sanders and T. Scholtz. First year students' understanding of the flow of control in recursive algorithms. *African Journal of Research in Mathematics, Science and Technology Education*, 16(3):348–362, 2012.

[91] Ian Sanders, Vashti Galpin, and T. Götschi. Mental models of recursion revisited. *SIGCSE Bulletin*, 38(3):138–142, June 2006.

[92] A. Savinainen and P. Scott. The force concept inventory: a tool for monitoring student learning. *Physics Education*, 37(1):45–52, January 2002.

[93] T. L. Scholtz and I. Sanders. Mental models of recursion: investigating students' understanding of recursion. In *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*, ITiCSE '10, pages 103–107, 2010.

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

[95] C. A. Shaffer. Practical introduction to data structures and algorithm analysis, 2004.

[96] C. A. Shaffer, V. Karavirta, A. Korhonen, and T. L. Naps. OpenDSA: Beginning a community hypertextbook project. In *Proceedings of the eleventh Koli Calling International Conference on Computing Education Research*, pages 112–117, November 2011.

[97] R. Sooriamurthi. Problems in comprehending recursion and suggested solutions. In *Proceedings of the sixth annual conference on Innovation and technology in computer science education*, ITiCSE '01, pages 25–28, 2001.

[98] R. Sooriamurthi. Problems in comprehending recursion and suggested solutions. In *Proceedings of the sixth annual conference on Innovation and technology in computer science education*, ITiCSE '01, pages 25–28, 2001.

[99] J. Stasko, A. Badre, and C. Lewis. 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, pages 61–66, 1993.

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

[101] L. Stern and L. Naish. Visual representations for recursive algorithms. *SIGCSE Bulletin*, 34(1):196–200, 2002.

[102] R. A Streveler, B. M. Olds, R. L. Miller, and M. A. Nelson. Using a delphi study to identify the most difficult concepts for students to master in thermal and transport science. In *Proceedings of the Annual Conference of the American Society for Engineering Education*, 2003.

[103] A. Taherkhani, L. Malmi, and A. Korhonen. Algorithm recognition by static analysis and its application in students' submissions assessment. In *Proceedings of the eighth International Conference on Computing Education Research*, Koli '08, pages 88–91, 2008.

[104] C. Taylor, D. Zingaro, L. Porter, K. C. Webb, C. B. Lee, and M. Clancy. Computer science concept inventories: past and future. *Computer Science Education*, 24(4):253–276, 2014.

[105] J. Tessler, B. Beth, and C. Lin. Using cargo-bot to provide contextualized learning of recursion. In *Proceedings of the ninth annual international ACM conference on International computing education research*, ICER '13, pages 161–168, 2013.

[106] A. E. Tew and M. Guzdial. The FCS1: a language independent assessment of CS1 knowledge. *SIGCSE Bulletin*, (6):111–116, 2011.

[107] A. E. Tew and M. Guzdial. Investigating factors of student learning in introductory courses. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, 2013.

[108] G. Thorburn and G. Rowe. PASS: an automated system for program assessment. *Computer Education*, 29(4):195–206, December 1997.

[109] N. Truong, P. Roe, and P. Bancroft. Static analysis of students' Java programs. In *Proceedings of the Sixth Australasian Conference on Computing Education*, pages 317–325, 2004.

[110] S. Tung, C. Chang, W. Wong, and J. Jehng. Visual representations for recursion. *International Journal of Human-Computer Studies*, 54(3):285–300, March 2001.

[111] J. D. Ullman and A. V. Aho. *Foundations of Computer Science Computer.* Science Press, 1992.

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

[113] T. Vilner, E. Zur, and J. Gal-Ezer. Recursive thinking in CS1? In *Proceedings of the ACM International Middleware Conference*, pages 189–197, 2008.

[114] K. C. Webb and C. Taylor. Developing a pre-and post-course concept inventory to gauge operating systems learning. *SIGCSE Bulletin*, pages 103–108, 2014.

[115] S. Wiedenbeck. Learning recursion as a concept and as a programming technique. *SIGCSE Bulletin*, 20(1):275–278, February 1988.

[116] D. Wilcocks and I. Sanders. Animating recursion as an aid to instruction. *Computers and Education*, 23(3):221 – 226, 1994.

[117] M. Wirth. Introducing recursion by parking cars. *SIGCSE Bulletin*, 40(4):52–55, November 2008.

[118] C. Wu, N. B. Dale, and L. J. Bethel. Conceptual models and cognitive learning styles in teaching recursion. *SIGCSE Bulletin*, 30(1):292–296, March 1998.

[119] C. Wu, G. C. Lee, and J. M. Lin. Visualizing programming in recursion and linked lists. In *Proceedings of the 3rd Australasian conference on Computer science education*, ACSE '98, pages 180–186, 1998.

[120] S. Xu and Y. S. Chee. Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Transactions on Software Engineering*, 29(4):360–384, 2003.

[121] F. Yang. Another outlook on linear recursion. *SIGCSE Bulletin*, 40(4):38–41, November 2008.

[122] Daniel Zingaro, Andrew Petersen, and M. Craig. Stepping up to integrative questions on CS1 exams. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 253–258, 2012.

Appendix A

RecuTutor Exercises

A.1 Exercises Detailed Information

Table A.1: Writing Practice Exercises Detailed Information

Exercise Name	**Description**	**Module**	**Recursion Type**	**Misconceptions Covered**
largest [22]	returns the largest number in a given array	Code Completion Set 1	Tail	BCwrite
print [22]	concatenates the values in an array and returns it in a String	Code Completion Set 1	Tail	BCbeforeRecursiveCase, RCwrite
mult	returns the multiplication of two given numbers	Code Completion Set 1	Tail	BCactionReturnConstant BCcheckAganistConstant, BCwrite
GCD	computes GCD of two numbers	Code Completion Set 1	Tail	BCactionReturnConstant BCcheckAganistConstant, BCwrite
log	computes log of a given number to a given base	Code Completion Set 2	Tail	BCactionReturnConstant BCwrite, RCwrite
power [70]	returns power of given number to a given power	Code Completion Set 2	Tail	RCwrite

mystery	returns cumulative sum from 1 to a given number	Code Completion Set 2	Tail	RCwrite
addodd	sums all the odd numbers less than a given number	Code Completion Set 3	Binary	RCwrite
sum	sums all the numbers between two given numbers	Code Completion Set 3	Tail	RCwrite
sumOfDigits	sums the digits of a given number	Code Completion Set 3	Tail	RCwrite
getDigits [70]	counts the digits in given number	Code Completion Set 4	Tail	RCwrite
countChr	counts the number of As in a given string	Code Completion Set 4	Tail	RCwrite
Fibonacci	computes Fibonacci of a given number	Harder Code Completion	Binary	RCwrite , BCbeforeRecursiveCase
decibinary [70]	computes the binary equivalent of a given decimal	Harder Code Completion	Binary	RCwrite
recursiveMin	computes the minimum of a given array	Harder Code Completion	Tail	RCwrite
isReverse	returns if the given strings are the reverse of each others	Harder Code Completion	Binary	BCwrite, BCcheckAganistConstant
Prime	returns if the given number is prime or not	Harder Code Completion	Tail	BCwrite

minOps [32], [113]	takes two positive integers X and Y as its input where X less than Y, and outputs the minimal number of invocations of the operations +1 and 2 that are required to obtain Y from X.	Writing Set 1	Binary	BCwrite,RCwrite
numOfPaths [32], [113]	counts the number of different ways to reach a basketball score	Writing Set 1	Binary	BCwrite,RCwrite
countInversions [32]	counts the number of inversions in a list of numbers	Writing Set 1	Binary	BCwrite,RCwrite, GlobalVariable
isBalanced [70]	returns if the bracketing operators in a given string is balanced	Writing Set 1	Tail	BCwrite,RCwrite
checkPalindrome [70]	returns if a given string read the same forwards as backwards	Writing Set 2	Tail	BCwrite,RCwrite
reverseString	returns the reverse of a given string	Writing Set 2	Tail	BCwrite,RCwrite,, GlobalVariable
issubsetsum	returns if a subset of a given set of numbers sums to a target number	Writing Set 2	Binary	BCwrite, RCwrite, GlobalVariable
removeDuplicates	removes all the duplicates in a given array	Writing Set 2	Binary	BCwrite,RCwrite, GlobalVariable

stackReversal	reverses a given stack without using any data structures	Writing Set 3	Linear	BCwrite,RCwrite, GlobalVariable
pascal	returns the value in a given position in the Pascal triangle	Writing Set 3	Binary	BCwrite,RCwrite
cannonball	returns the number of cannonballs in a given height	Writing Set 3	Tail	BCwrite,RCwrite
ismeasurable	returns if a certain target weight can be measured or not	Writing Set 3	Binary	BCwrite,RCwrite, GlobalVariable

Table A.2: Tracing Practice Exercises Detailed Information

Exercise Name	**Description**	**Module**	**Recursion Type**	**Misconceptions Covered**
mystryab	what is the return of the function when the initial call will hit the base case	Code Tracing Set 1	Tail	BCevaluation
mystryabinf	what is the return of the function when it leads to infinite recursion	Code Tracing Set 1	Tail	InfiniteExecution, BCevaluation
dosmthng5 [22]	what is the return of the function when the backward flow is executed	Code Tracing Set 1	Linear	BCevaluation, BFneverExecute,BFexecuteBefore
mystrynad	what is the return of the function when it is executed	Code Tracing Set 1	Tail	BCevaluation
mystryrsltn	what is the return of the function when it is executed	Code Tracing Set 1	Tail	BCevaluation
mystryexecn	what is the return of the function when it is executed	Code Tracing Set 1	Tail	BCevaluation
astrsksPrntr	what is the number of asterisks printed when the function is executed	Code Tracing Set 1	Binary	BCevaluation, BFneverExecute,BFexecuteBefore and multiple recursive calls
mystryn	what is the return of the function when it is executed	Code Tracing Set 1	Binary	BCevaluation and multiple recursive calls

mystryfkn	what is the return of the function when it is executed	Code Tracing Set 1	Binary	BCevaluation and multiple recursive calls
dosmthng5f	what is the return of the function when it is executed	Code Tracing Set 1	Tail	BCevaluation
rsltn	what is return of the function when it is executed	Code Tracing Set 2	Tail	BCevaluation
foox	how many times a recursive call is executed	Code Tracing Set 2	Tail	BCevaluation
strngrecur	when will the function terminate without errors	Code Tracing Set 3	Tail	BCevaluation
wrtwthcmms	what is the number that makes the function is executes properly	Code Tracing Set 3	Tail	BCevaluation
wrtwthcmmserrfx	how to fix the error in the previous exercise	Code Tracing Set 3	Binary	BCevaluation, BCwrite and multiple recursive calls
funcxy	what is the return of the function when it leads to infinite recursion	Code Tracing Set 4	Tail	BCevaluation, InfiniteExecution
printString	what is done by a given function	Code Tracing Set 5	Linear	BCevaluation, BFneverExecute,BFexecuteBefore
fnctnxn	what is done by the given function	Code Tracing Set 5	Linear	BCevaluation, BFneverExecute,BFexecuteBefore

sprwrtvrtcl	what are the values handled by the base case	Code Tracing Set 6	Binary	BCevaluation, BFneverExecute, BFexecuteBefore, BCbeforeRecursiveCase, multiple recursive calls
sprwrtvrtclcls	choose the number that leads to the most recursive calls	Code Tracing Set 6	Binary	BCevaluation, BFneverExecute, BFexecuteBefore, BCbeforeRecursiveCase, multiple recursive calls
mystrynumb	what is the return of a given function	Code Tracing Set 6	Tail	BCevaluation

A.2 Examples Detailed Information

Table A.3: Examples Detailed Information

Example Name	**Description**	**Module**	**Recursion Type**	**Misconceptions Covered**
multiply	multiply two numbers	Introduction	Tail	BCevaluation
sum	sum the numbers in a given array	Writing a recursive function	Tail	BCevaluation, BCwrite, RCwrite, BCbeforeRecursiveCase
sum	sum the numbers in a given array	Writing a recursive function	Tail	BCevaluation, BCwrite, RCwrite, BCbeforeRecursiveCase
Prime	returns if the given number is prime or not	Writing a more sophisticated recursive function	Tail	BCwrite
issubsetsum	returns if a subset of a given set of numbers sums to a target number	Writing a more sophisticated recursive function	Binary	RCwrite and multiple recursive calls
numOfPaths	counts the number of different ways to reach a basketball score	Writing a more sophisticated recursive function	Binary	BCwrite,RCwrite and multiple recursive calls
factorial	computes the factorial of a given number	Tracing recursive code	Tail	BCevaluation
Domino PrintOneToN	print the values from 1 to N	Tracing recursive code	Linear	BCevaluation, BFneverExecute, BFexecuteBefore
Domino NumOfDigits	counts the number of digits in a given number	Tracing recursive code	Linear	BCevaluation, BFneverExecute, BFexecuteBefore
Towers of Hanoi [64], [22], [41]	shows the towers of Hanoi solution	Tracing recursive code	Binary	BCevaluation, BFneverExecute, BFexecuteBefore

A.3 Exercises Item Analysis

Table A.4: Writing Practice Exercises Item Analysis

Exercise Name	**Difficulty Index**	**Discrimination Index**
largest	-0.594	0.671
print	0.476	0.463
mult	-2.778	0.475
GCD	-2.304	0.415
log	-0.450	0.431
power	-1.508	0.570
mystery	-3.158	0.317
addodd	-1.309	0.335
sum	-0.970	0.833
sumOfDigits	-0.756	0.834
getDigits	-1.285	1.949
countChr	-1.285	1.949
Fibonacci	-0.855	0.925
decibinary	1.819	0.411
recursiveMin	-0.299	1.072
isReverse	0.530	0.958
Prime	0.317	0.697
minOps	1.054	1.307
numOfPaths	1.289	1.111
countInversions	1.331	1.234
isBalanced	0.695	1.625
checkPalindrome	0.464	2.903
reverseString	0.557	1.716
issubsetsum	1.373	1.933
removeDuplicates	1.157	1.386
stackReversal	2.902	0.950
pascal	0.778	2.210
cannonball	0.272	2.237
ismeasurable	2.653	1.481

Table A.5: Tracing Practice Exercises Item Analysis

Exercise Name	**Difficulty Index**	**Discrimination Index**
Tracing Summary 1	-0.522	0.990
Tracing Summary 2	-1.155	2.134
Tracing Summary 3	-1.655	0.809
Tracing Summary 4	-1.655	1.024
Tracing Summary 5	-0.692	4.674
Tracing Summary 6	2.724	10.076

Appendix B

BTRecuTutor Exercises

B.1 Exercises Detailed Information

Table B.1: Writing Practice Exercises Detailed Information in Recursion in Binary Trees Tutorial

Exercise Name	**Description**	**Category**
Increment	increment all the values of the nodes of a binary tree by one	Local
Count Leaf	count the number of Leaf Nodes in a binary Tree	Collect and return
Depth	get the depth of a binary Tree	Collect and return
Check Value	check on the existence of a given value in a binary tree	Collect and return
Count Value	count the number of existences of a given value in a binary tree	Collect and return
Sum All	sum all the values of the nodes in a binary tree	Collect and return
Has Path Sum	check if any path from root to leaf has a given sum	Collect and return
Get Difference	get the difference between the values on the left sub-tree and the right sub-tree	Collect and return
Diameter	get the diameter of a given sub-tree	Collect and return
Check Sum	check if the for each node the sum of its children is equal to its value	Collect and return

Minimum	find the minimum in a binary search tree	Guided
Small Count	count the number of existences of a nodes values less than a given value in a binary search tree	Guided
Same Tree	check if the values in two given trees are the same	Multiple Trees
Swaps Trees	swap the values of two given trees	Multiple Trees
Structurally Identical Trees	check if two given trees are structurally identical	Multiple Trees
Mirror Trees	check if the values in the nodes of two given trees are mirrored	Multiple Trees

Appendix C

CS2114 Exam Questions

C.1 Pre-Test Questions

1. Given the following method:

   ```
   int Question1(int Q1_var1, int Q1_var2)
   {
    if (Q1_var2 == 1)
     return Q1_var1;
    else
     return Q1_var1 + Question1(Q1_var1, Q1_var2+1);
   }
   ```

   What is the returned value of the invocation Question1(2,3)? Either give a number, or if it will eventually lead to an infinite recursion just write "infinite recursion".
2. Given the following method:

   ```
   void Question2(int Q2_var1)
   {
    if(Q2_var1 > 0)
    {
     Question2((Q2_var1)-1);
     System.out.print(Q2_var1);
    }
   }
   ```

   What will be printed after the invocation Question2(7) ? Either give a sequence of numbers, or if it will eventually lead to an infinite recursion just write "infinite recursion".

3. Given the following method:

   ```
   int Question3( int x , int y)
   {
    if (x==0 || y==0 || x==y)
     return 1;
    else
     return Question3(x-1, y-1) + Question3(x-1, y);
   }
   ```

   What will be printed after the invocation mystery(4,2)? Either give a sequence of numbers, or if it will eventually lead to an infinite recursion just write "infinite recursion".
4. Write a recursive method int recursiveMax(int[] array, int index) that takes an array and an index (which is initially equal to zero) and return the largest integer in the array. Your routine must be recursive. Assume that the function will be called initially with index=0.

   ```
   int recursiveMax(int[] array, int index)
   {

   // implementation, remember recursive and base cases

   }
   ```

5. Write a recursive method boolean isSubsetSum(int set[], int n, int sum) that takes a set of integers, the number of integers and a target sum, your goal is to find whether a subset of those numbers adds up to the target sum. For example, given the set 3,7,1,8,-3 and the target sum 4, the subset 3,1 sums to 4. On the other hand, if the target is 2 then the result is false. It is only required to return true or false.

   ```
   boolean isSubsetSum(int set[], int n, int sum)
   {

   // implementation, remember recursive and base cases


   }
   ```

C.2 Post-Test Questions

1. Given the following method:

```
public void dosomething (int n)
{
   if ( n > 0 ) {
      System.out.print(n);
      dosomething( n-1 );
   }
}
```

What will be printed when `dosomething( 5 )` is executed? (Either write a sequence of numbers, or write infinite recursion.)

2. Given the following method:

```
public int func(int x, int y)
{
   if (y == 1)
      return x;
   else
      return x + func(x, y+1);
}
```

What will be printed when `func(2,3)` is executed? (Either write a number, or write infinite recursion.)

3. Read the following recursive method. Give the missing code such that this function when passed 2 numbers, will find the sum of all the integers between them. Example: given 1 and 4, the method should add 1+2+3+4 and return 10.

```
public int Sum(int a, int b)
{
   if (a == b)
      // Missing code
   else
      return Sum(a,b-1)+b;
}
```

When a space is needed, do NOT enter multiple spaces.

4- Write a recursive method int `BinaryToInt(String binary)` that takes a string with a binary number (it only has 1 and 0s) and returns the integer (numeric) representation of the number. Your routine must be recursive.

```
int BinaryToInt(String binary)
{
   // implementation, remember recursive and base cases
}
```

For example, the following code `System.out.println(BinaryToInt("1010"))` prints 10.

How to convert from binary to decimal: Each position in a binary number is valued at $2^{position}$. That is, the right most value is 2^0, the next right most is 2^1, etc.

Binary digit	1	0	1	0
Power of 2	2^3	2^2	2^1	2^0
Decimal value	8	4	2	1

Appendix D

Recursion Concept Inventory Questions

D.1 First iteration CI Questions

Backward Flow

1 Given the following code:

```
int function(int y) {
  if (y == 1)
    return 5;
  else {
    function(y - 1);
    y = y + 1;
    return 83;
  }
}
```

What will be returned when `function(2)` is executed? Write a number, or write "infinite recursion" if you think that this call will lead to infinite recursion.

Table D.1: Question item 1 Rubric

Answer	**Misconception**
83	**Correct**
5	**BFneverExecute**
6	**RCnoReturnRequired**
Infinite recursion	**BFexecuteBefore**
Other	?

2 Consider the following function.

```
void PrintArray(int[] A, int n) {
  if (n > 0) {
    PrintArray(A, n - 1);
    System.out.print(A[n]);
  }
}
```

What will be printed when `PrintArray(A, 5)` is executed, with array `A` initialized so that position `A[i]` stores value `i`? Write a sequence of numbers that will be printed, or write "nothing" if you think that it will print nothing. Write "infinite recursion" if you think that the call will lead to infinite recursion.

Table D.2: Question item 2 Rubric

Answer	**Misconception**
12345	**Correct**
1234	**BCevaluation**
01234	**BCevaluation**
012345	**BCevaluation**
54321	**BFexecuteBefore**
543210	**BFexecuteBefore and BCevaluation**
4321	**BFexecuteBefore and BCevaluation**
Nothing	**BFneverExecute**
Infinite recursion	**BCevaluation or ?**
Other	?

Infinite recursion

3. Consider the following function.

```
int mystery(int x) {
  if (x > 0)
    return 8;
  else
    return 2 + mystery(x - 1);
}
```

What value will be returned when `mystery(0)` is executed? Write a number, or write "infinite recursion" if you think that the call will lead to infinite recursion.

Table D.3: Question item 3 Rubric.

Answer	**Misconception**
Infinite recursion	**Correct**
8	**InfiniteExecution**
2 or 10, 12, 14, etc.	**BCevaluation**
Other	?

Recursive call

4. The following code leads to infinite recursion when called as `function(3, 2)`:

```
1  int function(int x, int y) {
2    if (x == y)
3      return y;
4    else
5      return function(x + 1, y);
6  }
```

Pick ONE line that you think is the cause of the infinite recursion and write a replacement, so that this replacement will fix the infinite recursion.

Table D.4: Question item 4 Rubric.

Answer	**Misconception**
Line 5 **return function(x - 1 , y)**	**Correct**
Line 2 **x ! = y**	**Correct**
Line 5 **return function(x , y+1)**	**Correct**
Line 5 **function(x - 1 , y)**	**RCnoReturnRequired**
Line 2 **x > any positive number or x == any number >= 3**	**Correct but maybe BCactionReturnConstant**
Line 2 **x < any positive number**	**BCwrite and BCactionReturnConstant**
Line 5 **function(x + any positive number)**	**RCwrite**
Line 5 **return y- any positive number**	**BCevaluation**
Line 5 **return any constant value**	**BCcheckAganistConstant**
Other	?

5. Given the following incomplete code:

```
int SumTo(int k)
{
  if (k > 0)
    // missing line;
  else
    return 0;
}
```

Write something to replace the line `// missing line` so that when given a number k, `SumTo` will return a cumulative sum of the values from 1 to k. For example, 15 will be returned when `SumTo(5)` is called, 21 when when `SumTo(6)` is called, and so on.

Table D.5: Question item 5 Rubric.

Answer	**Misconception**
return k + SumTo(k - 1)	**Correct**
return SumTo(k - 1)	**RCwrite**
return k + SumTo(k + 1)	**RCwrite**
k + SumTo(k - 1)	**RCnoReturnRequired**
k + SumTo(k + 1)	**RCnoReturnRequired and RCwrite**
Any answer that has no recursive call	**BCbeforeRecursiveCase**
Other	?

6. The following incomplete code is meant to print the numbers going from `y` down to `x`, where `x` < `y`. For example, if `CountDown(3, 7)` is called then the following should be printed: 76543

```
void CountDown(int x, int y) {
  if (x <= y) {
    System.out.print(y);
    // missing recursive call
  }
}
```

Write a recursive call that should replace `// missing recursive call`.

Table D.6: Question item 6 Rubric.

Answer	Misconception
CountDown(x , y-1)	**Correct**
CountDown(x , y+1)	**RCwrite and BCevaluation**
return CountDown(x , y-1)	**RCreturnIsRequired**
return CountDown(x , y+1)	**RCreturnIsRequired and RCwrite and BCevaluation**
Any answer that has no recursive call	**BCbeforeRecursiveCase**
Other	?

Base case

7. Given the following two methods:

```
int function1(int x, int y) {
  if (x == 1)
    return y;
  else
    return function1(x-1, y) + y;
}

int function2(int x, int y) {
  if (x > 1)
    return function2(x-1, y) + y;
  else
    return y;
}
```

   What values are returned by the calls `function1(2,3)` and `function2(2,3)`? Write a number for each return value, or write "infinite recursion" if you think either will eventually lead to infinite recursion.

8. Given the following incomplete recursive method:

```
int Sum(int a, int b) {
 if ( //Missing Case// )
   //Missing Action//
 else
   return Sum(a, b-1)+ b;
}
```

Table D.7: Question item 7 Rubric

Answer	**Misconception**
6 and 6	**Correct**
Two different values	**BCbeforeRecursiveCase**
The same value, but not 6	**BCevaluation or ?**
Infinite Recursion for both	**BCevaluation or ?**
Other	**?**

Write something to replace `//Missing Case//` and `//Missing Action//` so that when this recursive function is passed 2 numbers, it will return the sum of all the integers between them. For example, given 2 and 5, add 2 + 3 + 4 + 5 and return 14. If the two numbers are equal, then return that value.

Table D.8: Question item 8 Rubric.

Answer	**Misconception**
a==b and return a	**Correct**
a==b and return b	**Correct**
a==b and return constant	**BCactionReturnConstant**
A condition like a==constant or b==constant and return a	**BCwrite and BCcheckAganistConstant**
Other	**?**

Variables updating

9. The following function is intended to find the minimum value in an array.

```
int recursiveMin(int[] array, int index) {
  int min = array[0];
  if (index == 0)
    return min;
  else {
    if (array[index] < min)
      min = array[index];
    return recursiveMin(array, index-1);
  }
}
```

What will be returned by `recursiveMin` when the following lines are executed?

```
int [] array = {10, 20, 2, 30, 8};
int var= recursiveMin(array, array.length);
```

Write a number, or write “infinite recursion” if you think that the call will lead to infinite recursion.

Table D.9: Question item 9 Rubric

Answer	**Misconception**
10	**Correct**
2	**GlobalVariable**
8	**?**
Infinite Recursion	**?**
Other	**?**

Writing Question

10. Write a recursive function to compute x to the power y. Assumes that y is positive or zero and both x any y are integers.

D.2 Analysis of First Draft CI

Table D.10: Questions by Misconception and Difficulty

Misconception	**Questions**
BFneverExecute	**item 1, item 2**
BFexecuteBefore	**item 1, item 2**
InfiniteExecution	**item 3, item 4**
RCwrite	**item 4,item 5, item 6, item 10**
RCnoReturnRequired	**item 1, item 5, item 4,item 10**
RCreturnIsRequired	**item 5, item 6**
BCbeforeRecursiveCase	**item 5, item 7**
BCactionReturnConstant	**item 4, item 8**
BCcheckAganistConstant	**item 4, item 8**
BCwrite	**item 4, item 8, item 10**
BCevaluation	**item 2, item 3, item 4, item 6, item 7, item 10**
GlobalVariable	**item 9**

D.3 Second Iteration CI Questions

1 Given the following code:

```
int function(int y) {
  if (y == 1)
    return 5;
  else {
    function(y - 1);
    y = y + 1;
    return 83;
  }
}
```

What will be returned when `function(2)` is executed? Write a number, or write “infinite recursion” if you think that this call will lead to infinite recursion.

2 Given the following incomplete code:

```
int SumTo(int k)
{
  if (k > 0)
    // missing line;
  else
```

Table D.11: Misconceptions and Difficulties by Question

item 1	**BFneverExecute, RCnoReturnRequired, BFexecute-Before**
item 2	**BFexecuteBefore, BCevaluation, BFneverExecute**
item 3	**InfiniteExecution, BCevaluation**
item 4	**RCwrite, BCevaluation, BCcheckAganistConstant, BCactionReturnConstant, BCwrite, RCnoReturnRequired**
item 5	**RCwrite, RCreturnIsRequired, RCnoReturnRequired, BCbeforeRecursiveCase**
item 6	**RCwrite, RCreturnIsRequired, BCevaluation**
item 7	**BCbeforeRecursiveCase , BCevaluation**
item 8	**BCactionReturnConstant, BCwrite, BCcheckAganistConstant**
item 9	**GlobalVariable**
item 10	**RCwrite, RCnoReturnRequired, BCwrite, BCevaluation**

Table D.12: Question item 1 Rubric

Answer	**Misconception**
83	**Correct**
5	**BFneverExecute**
6	**RCnoReturnRequired**
Infinite recursion	**BFexecuteBefore**
583	**BCbeforeRecursiveCase**
Other	?

```
    return 0;
}
```

Write something to replace the line `// missing line` so that when given a number k, `SumTo` will return a cumulative sum of the values from 1 to k. For example, 15 will be returned when `SumTo(5)` is called, 21 when when `SumTo(6)` is called, and so on.

3 The following incomplete code is meant to print the numbers going from `y` down to `x`, where `x` < `y`. For example, if `CountDown(3, 7)` is called then the following should be printed: 76543

```
void CountDown(int x, int y) {
  if (x <= y) {
    System.out.print(y);
    // missing recursive call
```

Table D.13: Question item 2 Rubric.

Answer	Misconception
return k + SumTo(k - 1)	**Correct**
return SumTo(k - 1)	**RCwrite**
return k + SumTo(k + 1)	**RCwrite**
k + SumTo(k - 1)	**RCnoReturnRequired**
k + SumTo(k + 1)	**RCnoReturnRequired and RCwrite**
Any answer that has no recursive call	**BCbeforeRecursiveCase**
Other	?

```
  }
}
```

Write a recursive call that should replace `// missing recursive call`.

Table D.14: Question item 3 Rubric.

Answer	Misconception
CountDown(x , y-1)	**Correct**
CountDown(x , y+1)	**RCwrite and BCevaluation**
return CountDown(x , y-1)	**RCreturnIsRequired**
return CountDown(x , y+1)	**RCreturnIsRequired and RCwrite and BCevaluation**
Other	**?**

4. Given the following two methods:

```
int function1(int x, int y) {
  if (x == 1)
    return y;
  else
    return function1(x-1, y) + y;
}

int function2(int x, int y) {
  if (x > 1)
    return function2(x-1, y) + y;
  else
    return y;
}
```

What values are returned by the calls `function1(2,3)` and `function2(2,3)`? Write a number for each return value, or write "infinite recursion" if you think either will eventually lead to infinite recursion.

Table D.15: Question item 4 Rubric

Answer	**Misconception**
6 and 6	**Correct**
Two different values	**BCbeforeRecursiveCase**
The same value, but not 6	**BCevaluation or ?**
Infinite Recursion for both	**BCevaluation or ?**
Other	**?**

5. Given the following incomplete recursive method:

```
int Sum(int a, int b) {
 if ( //Missing Case// )
   //Missing Action//
 else
   return Sum(a, b-1)+ b;
}
```

Write something to replace `//Missing Case//` and `//Missing Action//` so that when this recursive function is passed 2 numbers, it will return the sum of all the integers between them. For example, given 2 and 5, add 2 + 3 + 4 + 5 and return 14. If the two numbers are equal, then return that value.

Table D.16: Question item 5 Rubric.

Answer	**Misconception**
a==b and return a	**Correct**
a==b and return b	**Correct**
a==b and return constant	**BCactionReturnConstant**
A condition like a==constant or b==constant and return a	**BCwrite and BCcheckAganistConstant**
a==b and return a+b	**?**
Other	**?**

6. Write a recursive function to search for a given value in a given array.

D.4 Analysis of Second Draft CI

Table D.17: Questions by Misconception and Difficulty

Misconception	**Questions**
BFneverExecute	**item 1**
BFexecuteBefore	**item 1**
RCwrite	**item 2, item 3, item 6**
RCnoReturnRequired	**item 1, item 6**
RCreturnIsRequired	**item 2, item 3**
BCbeforeRecursiveCase	**item 1, item 4**
BCactionReturnConstant	**item 5**
BCcheckAganistConstant	**item 5**
BCwrite	**item 2, item 3, item 6**
BCevaluation	**item 2, item 3, item 4, item 6**
GlobalVariable	**item 6**

Table D.18: Misconceptions and Difficulties by Question

item 1	**BFneverExecute, RCnoReturnRequired, BFexecute-Before, BCbeforeRecursiveCase**
item 2	**RCwrite, RCreturnIsRequired, BCevaluation**
item 3	**RCwrite, RCreturnIsRequired, BCevaluation**
item 4	**BCbeforeRecursiveCase , BCevaluation**
item 5	**BCactionReturnConstant, BCwrite, BCcheckAgan-istConstant**
item 6	**RCwrite, RCnoReturnRequired, BCwrite, BCevalua-tion,GlobalVariable**

Table D.19: Misconceptions and Questions Matrix

Misconception	**Q1**	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**
BFneverExecute	X					
BFexecuteBefore	X					
RCwrite		X	X			X
RCnoReturnRequired	X					X
RCreturnIsRequired		X	X			
BCbeforeRecursiveCase	X			X		
BCactionReturnConstant					X	
BCcheckAganistConstant					X	
BCwrite		X	X			X
BCevaluation		X	X	X		X
GlobalVariable						X

Appendix E

Recursion in Binary Tree Test Questions

E.1 Pre-test Questions

1. Write a recursive function named `bstMin` that, given the root to a Binary Search Tree (BST), returns a reference to the node that has the minimum value found in the passed tree. Function `bstMin` should visit as few nodes in the BST as possible. Function `bstMin` should have the following prototype:

   ```
   BinNode bstMin(BinNode root)
   ```

   The Correct Answer:

   ```
   BinNode bstMin(BinNode root){
    if (root == null)
      return null;

    if (root.left() == null)
      return root;

    return bstMin(root.left());
   }
   ```

2. Write a recursive function named `btCheckVal` that, given the root to a Binary Tree and value, returns true if there is a node in the given binary tree with the given value, and false otherwise. Function `btCheckVal` should have the following prototype:

   ```
   boolean btCheckVal(BinNode root , int value)
   ```

   The Answer:

Table E.1: Question item 1. Rubric.

Answer	**Misconception**
Solution that access root.right()	**BSTMinCheckRight**
Solution that does not access root.left()	**BSTMinNoCheckLeft**
Solution that does not check if root == null	**rootIsNotNull**
Solution that checks on root.isLeaf() as the base case	**rootIsLeaf**
Other	?

```
boolean btCheckVal(BinNode root , int value) {
  if (root == null)
    return false;
  else {
   if (root.element()== value)
      return true;
    else
      return btCheckVal(root.left(), value) || btCheckVal(root.right(), value);

    }
}
```

Table E.2: Question item 2. Rubric.

Answer	**Misconception**
Solution that access the values of the root's left or right children	**childCheckValue**
Solution that check if the root's left or right children is null	**childIsNull**
Solution that does not check if root == null	**rootIsNotNull**
Solution that checks on root.isLeaf() as the base case	**rootIsLeaf**
Other	?

E.2 Post-test Questions

3. Write a recursive function named `bstsmallCount` that, given the root to a Binary Search Tree (BST) and a value ''key" returns the number of nodes having values less than key. Function `bstsmallCount` should visit as few nodes in the BST as possible. Function `bstsmallCount` should have the following prototype:

   ```
   int  bstsmallCount(BinNode root , int key)
   ```

   The Answer:

```
int  bstsmallCount(BinNode root , int key) {
  if(root==null)
    return 0;

  if((Integer)root.element() < key)
    return 1 + bstsmallCount(root.left(), key) + bstsmallCount( root.right(), key)

  else
    return bstsmallCount(root.left(), key);
 }
```

Table E.3: Question item 3 Rubric.

Answer	**Misconception**
Solution that access root.right() in the else condition	**BSTMinCheckRight**
Solution that does not access root.left()	**BSTMinNoCheckLeft**
Solution that does not check if root == null	**rootIsNotNull**
Solution that checks on root.isLeaf() as the base case	**rootIsLeaf**
Solution that access the values of the root's left or right children	**childCheckValue**
Solution that check if the root's left or right children is null	**childIsNull**
Other	?

4. Write a recursive function named `btDepth` that, given the root to a Binary Tree the function finds the depth of the binary tree. The depth of a binary tree is the length of the path to the deepest node. An empty tree has a depth of 0, and a tree with a root node only has a depth of 1 and so on. Function `btDepth` should have the following prototype:

```
int btDepth(BinNode root)
```

The Answer:

```
int btDepth(BinNode root) {
  if (root == null)
    return 0;
  else {
    return 1 + Math.max(btDepth(root.left()), btDepth(root.right()));
  }
}
```

Table E.4: Question item 4 Rubric.

Answer	**Misconception**
Solution that check if the root's left or right children is null	**childIsNull**
Solution that does not check if root == null	**rootIsNotNull**
Solution that checks on root.isLeaf() as the base case	**rootIsLeaf**
Solution that misses a recursive call on the root.left() or the root.right()	**?**
Other	?

Appendix F

IRB Approval Letters

We received approval from the Virginia Tech Institutional Review Board for conducting our research on CS2114 and CS3114 students.

Office of Research Compliance
Institutational Review Board
North End Center, Suite 4120, Virginia Tech
300 Turner Street NW
Blacksburg, Virginia 24061
540/231-4606 Fax 540/231-0959
email irb@vt.edu
website http://www.irb.vt.edu

MEMORANDUM

DATE:	June 5, 2014
TO:	Cliff Shaffer, Jeremy V Ernst, N. Dwight Barnette, Susan Rodger
FROM:	Virginia Tech Institutional Review Board (FWA00000572, expires April 25, 2018)
PROTOCOL TITLE:	Collaborative Research: Assessing and Expanding the Impact of OpenDSA, an Open-Source, Interactive eTextbook for Data Structures and Algorithms
IRB NUMBER:	**14-623**

Effective June 5, 2014, the Virginia Tech Institution Review Board (IRB) Chair, David M Moore, approved the New Application request for the above-mentioned research protocol.

This approval provides permission to begin the human subject activities outlined in the IRB-approved protocol and supporting documents.

Plans to deviate from the approved protocol and/or supporting documents must be submitted to the IRB as an amendment request and approved by the IRB prior to the implementation of any changes, regardless of how minor, except where necessary to eliminate apparent immediate hazards to the subjects. Report within 5 business days to the IRB any injuries or other unanticipated or adverse events involving risks or harms to human research subjects or others.

All investigators (listed above) are required to comply with the researcher requirements outlined at:

http://www.irb.vt.edu/pages/responsibilities.htm

(Please review responsibilities before the commencement of your research.)

PROTOCOL INFORMATION:

Approved As:	**Expedited, under 45 CFR 46.110 category(ies) 5,7**
Protocol Approval Date:	**June 5, 2014**
Protocol Expiration Date:	**June 4, 2015**
Continuing Review Due Date*:	**May 21, 2015**

*Date a Continuing Review application is due to the IRB office if human subject activities covered under this protocol, including data analysis, are to continue beyond the Protocol Expiration Date.

FEDERALLY FUNDED RESEARCH REQUIREMENTS:

Per federal regulations, 45 CFR 46.103(f), the IRB is required to compare all federally funded grant proposals/work statements to the IRB protocol(s) which cover the human research activities included in the proposal / work statement before funds are released. Note that this requirement does not apply to Exempt and Interim IRB protocols, or grants for which VT is not the primary awardee.

The table on the following page indicates whether grant proposals are related to this IRB protocol, and which of the listed proposals, if any, have been compared to this IRB protocol, if required.

Date*	OSP Number	Sponsor	Grant Comparison Conducted?
05/30/2014	14163802	National Science Foundation	Compared on 06/05/2014

* Date this proposal number was compared, assessed as not requiring comparison, or comparison information was revised.

If this IRB protocol is to cover any other grant proposals, please contact the IRB office (irbadmin@vt.edu) immediately.

VirginiaTech

Office of Research Compliance
Institutional Review Board
North End Center, Suite 4120, Virginia Tech
300 Turner Street NW
Blacksburg, Virginia 24061
540/231-4606 Fax 540/231-0959
email irb@vt.edu
website http://www.irb.vt.edu

MEMORANDUM

DATE:	May 7, 2015
TO:	Cliff Shaffer, Jeremy V Ernst, N. Dwight Barnette, Susan Rodger
FROM:	Virginia Tech Institutional Review Board (FWA00000572, expires April 25, 2018)
PROTOCOL TITLE:	Collaborative Research: Assessing and Expanding the Impact of OpenDSA, an Open-Source, Interactive eTextbook for Data Structures and Algorithms
IRB NUMBER:	**14-623**

Effective May 7, 2015, the Virginia Tech Institution Review Board (IRB) Chair, David M Moore, approved the Continuing Review request for the above-mentioned research protocol.

This approval provides permission to begin the human subject activities outlined in the IRB-approved protocol and supporting documents.

Plans to deviate from the approved protocol and/or supporting documents must be submitted to the IRB as an amendment request and approved by the IRB prior to the implementation of any changes, regardless of how minor, except where necessary to eliminate apparent immediate hazards to the subjects. Report within 5 business days to the IRB any injuries or other unanticipated or adverse events involving risks or harms to human research subjects or others.

All investigators (listed above) are required to comply with the researcher requirements outlined at:

http://www.irb.vt.edu/pages/responsibilities.htm

(Please review responsibilities before the commencement of your research.)

PROTOCOL INFORMATION:

Approved As:	**Expedited, under 45 CFR 46.110 category(ies) 5,7**
Protocol Approval Date:	**June 5, 2015**
Protocol Expiration Date:	**June 4, 2016**
Continuing Review Due Date*:	**May 21, 2016**

*Date a Continuing Review application is due to the IRB office if human subject activities covered under this protocol, including data analysis, are to continue beyond the Protocol Expiration Date.

FEDERALLY FUNDED RESEARCH REQUIREMENTS:

Per federal regulations, 45 CFR 46.103(f), the IRB is required to compare all federally funded grant proposals/work statements to the IRB protocol(s) which cover the human research activities included in the proposal / work statement before funds are released. Note that this requirement does not apply to Exempt and Interim IRB protocols, or grants for which VT is not the primary awardee.

The table on the following page indicates whether grant proposals are related to this IRB protocol, and which of the listed proposals, if any, have been compared to this IRB protocol, if required.

Date*	OSP Number	Sponsor	Grant Comparison Conducted?
05/30/2014	14163802	National Science Foundation	Compared on 06/05/2014

* Date this proposal number was compared, assessed as not requiring comparison, or comparison information was revised.

If this IRB protocol is to cover any other grant proposals, please contact the IRB office (irbadmin@vt.edu) immediately.

Appendix G

Invitation Emails

G.1 CS 2144 Students Invitation Email

Hello, I am a PhD student in CS department. My research is on building tutorials to teach hard programming skills. I need your help on a study on recursion misconceptions. I need to interview you at my office in Torgerson 2000. The interview will not take more than half an hour. It is completely voluntary. Please let me know if you will be able to come and when? Thanks,
–
Sally Hamouda
PhD Candidate
Department of Computer Science
Virginia Tech

G.2 CS 3114 Students Invitation Email

Subject Name– I am looking to interview a small number of students in CS3114 this semester regarding various topics in the course that students tend to find difficult. I am hoping that you will agree to meet with me and my graduate student to discuss the presentation of material in the course and what topics you find easy or hard. I expect the interview to take a bit less than an hour. If you are willing to do this, when would be good times that you could meet with us after Thanksgiving break? (This could be either during the last week and a half of class, or during finals week.) Thanks in advance!

–
Cliff Shaffer, Professor
Department of Computer Science

Virginia Tech, Blacksburg, VA 24061

Appendix H

Instructors Interviews

H.1 Instructor 1

The instructor has been teaching classes at Virginia Tech that involve recursion for 30 years. He said that we should re-order the topics and reformulate them as follows:

- Recursive calls.
- Infinite recursion.
- Limiting case.
- Variables updating.
- Ascent flow and Descent flow
- Confusion with loop structure (least important).

He said that the Confusion with loop structure is the least important topic and can be taken of the list.

H.2 Instructor 2

The instructor has been teaching classes at Virginia Tech that involve recursion for 17 years. He did not like the terms "active flow" and "passive flow", but did not have a suggestion for alternatives. But we reached consensus on the alternate terms "ascent flow" and "descent flow". He said that confusion with loop structure is not an important misconception and should be taken off the list. He stated that infinite recursion is a symptom of misconception of either not knowing how to formulate the base case or how to make the recursive call. He suggested re-ordering the topics and reformulating them as:

- Selecting between the recursive case and the base case.
- Formulating the recursive call.

- Writing more than one recursive call.
- Parameter updating.
- Differentiating between Ascent flow and Descent flow
- Base case formulation.

Appendix I

Student Interviews

I.1 CS 2114 Interviews

I.1.1 Interview Script

At the beginning of the interview the students were asked the following questions:

- What is your confidence level about reading or tracing a recursive function?
- What is your confidence level about writing a recursive function?
- What are the problems that you have on recursion?

The students then were asked to solve the following 6 programming questions in half an hour:

1. Given the following recursive function that misses a recursive call. Write down the missing recursive call such that this function returns the sum of the integers from 1 to n.

```
int sum(int n)
{
 if (n == 1)
  return 1;
 else
  return //<<Missing a Recursive call>>
}
```

2. Given the following recursive function that misses a recursive call. Write down the missing recursive call such that this function prints the values in an array named list. The values must appear one per line in order of increasing subscript.

```
void print(String[] list, int index) {
      //<<Missing a Limiting case>>
          System.out.println(list[index]);
     print(list, index+1);
}
```

3. Consider the following function:

   ```
   int mystery(int a, int b) {
     if (b==1)
       return a;
     else
       return a + mystery(a, b-1);
   }
   ```

   What is the return of calling mystery(2,0)?
4. Consider the following code:

   ```
   public void dosomething (int n) {
    if(n>0) {
      dosomething(n-1);
      System.out.print(n);
     }
   }
   ```

   What will be printed when "dosomething(5)" is called? (Either write a sequence of numbers, or write "infinite recursion".)

5. Given the following code:

   ```
   public int exec(int n){
    if (n == 0)
     return 0;
   else
     return n + exec(n - 1);
   }
   ```

   What is the value that will be returned by the method call exec(5)? (Either write a number, or write "infinite recursion".)

6. Given the following:

```
int mystery (int[] numbers, int index) {
 if(index==numbers.length-1) {
  return numbers[index];
 }
 else if(numbers[index] > numbers[index+1]) {
  numbers[index+1] = numbers[index];
}
 return  mystery(numbers,index+1);
}
```

If initially numbers= 5, 9 , 20 , 2, 3 ,12 and index=0 What will be the value returned by this mystery function and what will be the value of index at the time of the last return?

7. Write a recursive function to compute the factorial of a positive number n.

After solving the questions, the students were asked to answer the following questions:

- What are the questions that you struggled with?
- What is your opinion on the questions?
- What is your opinion on your responses?
- Do you feel more or less confident about your recursion knowledge after solving those questions?

I.1.2 Subject Responses

Subject 1

The first student had solved most of the questions correctly from the first trial. However, answering the last question took a long time and the student needed a hint. She thought that the main thing that she struggled with during solving the questions was figuring out what the given function was doing, and that only the last question was difficult. She said that the term “limiting case” was confusing as she is used more to the term “base case”. She said that she was a bit more confident about her recursion knowledge after solving the exercises.

Subject 2

The second student was less confident about reading or tracing a recursive function and more confident about writing a recursive function. She felt that her main problem with recursion was to trace a function’s behavior. She felt that she was confident about her knowledge on loops because she had had enough practice, but she did not have the same practice on recursion. She took a relatively long time to solve the problems. She appeared confused on the problems where there was code after the recursive call. She believed that the function stops/returns after the recursive call and nothing would be executed after the recursive call (a clear confusion about passive flow). She got confused also about how to write the base case and differentiating it from the recursive case. She solved almost 70 percent of the questions correctly after deep thinking. She solved the last tracing question incorrectly because she got confused about how the base case was executed. She solved the writing code question correctly. The only thing that confused her regarding the writing question was that she could not remember whether the mathematical definition of factorial applies for the negative numbers or not. Once given the answer to this, she solved the question easily. She struggled most with the first, third, and last tracing question. She said that she felt more confident about her recursion knowledge after solving the programming questions.

I.2 CS 3114 Interviews

I.2.1 The interview questions

1. How confident are about writing recursive functions?
2. How confident are you about writing recursive programs related to binary trees and traversals?
3. What was the reason for the wrong answer on the recursion in binary tree mid-term Question?
4. Do you think that you have learned more about writing recursive functions since you took the midterm?
5. Do you think that you could now write this function correctly?
    - If yes, how would you figure out a fix?
    - If no, why could not you figure out a fix?
6. What do you think could help you to better understand the topic of recursive tree functions?
7. Do you have any suggestions on enhancing the presentation of the binary trees chapter in OpenDSA?

I.2.2 Subjects Responses

Subject 1

Student usually uses OpenDSA to prepare for the midterms by reading the chapter and then re-reading it one or two days before the midterm.

He thinks that understanding data structures is harder than understanding sorting.

He is generally comfortable with recursion. We were asking the students how could he solve the programming exercise on OpenDSA on binary trees that asks to count the leaf nodes in a given tree. The student could not remember how could he approaches it.

By looking on his attempts on the recursive tree traversal exercise in the database we have found that he had three attempts until he got the correct answer. However, his answer is doing un-necessary checks that could be avoided. In every attempt, he was trying to modify his code so that he can fix the errors that appears to him. His answer to the OpenDSA programming exercise shows that he is not very good in formulating the recursive case, nor the recursive call.

He thinks that OpenDSA exercises about binary trees are too easy. He thinks that the presentation of the binary trees in OpenDSA should be enhanced by showing real applications. He suggested that OpenDSA can have more difficult code writing exercises on binary trees and traversals.

After showing him his mid-term answer he thinks that the main reason for not getting the correct answer for the binary tree exercise is not writing the base case correctly. But he was not sure how to fix his answer. He did not have any suggestions about how could he enhance his understanding to binary trees because he says that he did not study it well enough.

Subject 2

She uses OpenDSA to study for the mid-terms and uses some of the OpenDSA examples, definitions, data structures usages in her cheat sheet for the midterms. This subject had a year gap between taking CS2114 and CS3114 for a reason related to a family emergency and that's why she used to work alone in the projects. She thinks that gap affect her performance so much in the 3114 class specially in the programming skills. Her first language is not English that's why she thinks that she spend more time than her class mates in reading and understanding the material and the terminologies. She said that C2114 is not doing a good job in preparing students to 3114 and their is a gap between them. She said that recursion is not well covered in 2114 and that's why she struggled in 3114 on the topics that are related to recursion.

Her confidence level about writing recursive functions is 4 on a scale from 1 to 10. She said that the practice exercises on trees and traversals are too easy. She has made 15 attempts until getting the answer correct in the programming exercise. She said she depended on the feedback she got from the programming exercise editor in OpenDSA to fix her errors and also she have looked up the internet to find out how to fix her problems.

When she have seen her midterm question she could not figure out why her answer was wrong and how to fix it. She said she learned more about recursion after the first midterm. She suggested that OpenDSA should have more practice and visualizations for the tree traversal topic to help student understand it better.

Subject 3

The subject uses OpenDSA and Google to study for the mid-terms. He generally likes to have multiple sources to study from. He is pretty confident in writing general recursive functions and recursive functions on trees. He thinks he is not a good test taker and that's why he missed up his answer to the binary trees question in the mid-term as he left that question for the last 5 minutes. He could figure out his problem in the answer of the binary tree mid-term question. He thinks that recursion is easy. He thinks that adding more visualizations that shows how a recursive function works by going through some examples will be beneficial in understanding recursion. He suggests also adding more programming exercises questions with different styles and tasks. He said that he uses a stack to trace how a recursive function works. He is generally avoiding using recursion in the problems that can be solved in another way. He has two years of programming experience and he thinks that OpenDSA adds to him the good part of knowing how recursion works. He says that OpenDSA is a very good system to learn from. He found that he had a problem in binary tree traversals in the recursion pre-test and mid-term1 although he thinks that he doesn't have any problems. He had 5 attempts in the programming exercise of the tree traversal in OpenDSA until getting the correct answer.

Endorses the idea of more programming exercises, and making the exercises more strict in the solution quality.

Subject 4

He uses OpenDSA for preparing for the mid-term. He uses Google as a studying resource but not for preparing for the mid-term. He thinks that OpenDSA helped him a lot. His understanding to recursion enhanced a lot due to the CS3114 in class explanantions. He got from the class that in order to understand recursion you need to understand well how the base case is working and avoid thinking about the details of the recusrion. He thinks that recursion was made much simpler in the CS3114 class. He is confident in writing recursive functions. When he was given his mid-term question he was able to fix it easily as he learnt more about recursion after that mid-term. He thinks that the current programming exercise on binary tree traversals is too easy and the reason why he made 8 attempts is that he misread it and then when he read it correctly it worked jsut fine. He thinks that adding more complicated programming exercises would help better understanding binary tree traversals.

He feels comfortable about recursion. He mentioned that the trigger to understanding was the explanation of recursion in class. In particular, he cited focusing on base cases and simplifying what you pay attention to in writing the recursive function (not looking at too many nodes).

Appendix J

Informed Consent Form

VIRGINIA POLYTECHNIC INSTITUTE AND STATE UNIVERSITY

Informed Consent for Participants
in Research Projects Involving Human Subjects

Title of Project: ___Tutorials for Teaching Recursion and Pointers____________

Investigator(s):	**____Clifford Shaffer_____**	**__shaffer@cs.vt.edu_**
	Name	E-mail / Phone number
	____Sally Hamouda_________	**__sallyh84@vt.edu__**
	Name	E-mail / Phone number

I. Purpose of this Research Project

The purpose of this study is to identify the common misconceptions the students have on recursion. The student should be enrolled in the CS2114 class in the current semester.

II. Procedures

Should you agree to participate, you will be asked to participate in a 30-minute audio-recorded interview. At the beginning of the interview you will be asked some general questions about your confidence level on recursion then you will be asked to solve six questions about recursion. We will ask you to say what are you thinking about while solving the questions. At the end of the interview, you will be asked to evaluate the questions and your responses.

III. Risks

There are no risks from participating in this study.

IV. Benefits

No promise or guarantee of benefits has been made to encourage you to participate.

V. Extent of Anonymity and Confidentiality

Your responses are completely anonymous. However, it will be used anonymously in publications from this study.

The Virginia Tech (VT) Institutional Review Board (IRB) may view the study’s data for auditing purposes. The IRB is responsible for the oversight of the protection of human subjects involved in research.

VI. Compensation

Your participation is completely voluntary.

VII. Subject's Consent

I have read the Consent Form and conditions of this project. I have had all my questions answered. I hereby acknowledge the above and give my voluntary consent:

____ Date__________
Subject signature

____
Subject printed name

VIII. Freedom to Withdraw

It is important for you to know that you are free to withdraw from this study at any time without penalty. You are free not to answer any questions that you choose or respond to what is being asked of you without penalty.

Please note that there may be circumstances under which the investigator may determine that a subject should not continue as a subject.

Should you withdraw or otherwise discontinue participation, you will be compensated for the portion of the project completed in accordance with the Compensation section of this document.

IX. Questions or Concerns

Should you have any questions about this study, you may contact one of the research investigators whose contact information is included at the beginning of this document.

Should you have any questions or concerns about the study's conduct or your rights as a research subject, or need to report a research-related injury or event, you may contact the VT IRB Chair, Dr. David M. Moore at moored@vt.edu or (540) 231-4991.