A Structured Approach to Teaching Recursion Using Cargo-Bot

Elynn Lee Victoria Shan Bradley Beth Calvin Lin
The University of Texas at Austin
Department of Computer Science
2317 Speedway, Stop D9500
Austin, TX 78712
{elynnlee, vshan, bbeth, lin}@cs.utexas.edu

ABSTRACT

Recursion is a notoriously difficult concept to learn. This paper presents a structured approach to teaching recursion that combines classroom lectures and self-paced interaction with Cargo-Bot, a video game in which users solve puzzles using a simple visual programming language. After mapping Cargo-Bot games to a set of learning goals, we devise a lesson plan that uses Cargo-Bot game playing to scaffold key concepts used in writing recursive Java programs. We empirically evaluate our approach using 204 undergraduates enrolled in a CS2 course, and we show strong statistical evidence that our approach improves student learning of recursion over traditional lecture-based instruction alone.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Recursion; K.3.2 [**Computer and Information Science Education**]: Computer Science Education

Keywords

Education; recursion; video games

1. INTRODUCTION

Recursion is a fundamental concept in computer science that novices often struggle to understand [14, 15, 22]. In particular, students often fail to understand the *passive flow* of recursion [22], which is the backward flow of control that can take place after reaching the base case. While there is considerable prior work in teaching recursion [1, 3, 6–12, 16–21, 23–25, 27–29], there exist few controlled empirical studies. Chaffin et al. [2] and Hulsizer [13] demonstrate statistically significant results in experiments with fewer than 17 students, but neither compares the results against a controlled baseline.

To date, the largest controlled empirical study [26] leverages a video game, Cargo-Bot, in which players write visual programs to solve puzzles involving cranes and boxes; procedure calls are the only construct for repetition, so the authors posit that this game can contextualize the learning of recursion. In their study involving 47 magnet school students taking AP Computer Science A, the authors find that an experimental group—which first plays Cargo-Bot for 70 minutes and then receives 50 minutes of direct instruction—sees a statistically significant improvement in assessment scores when compared against a control group—which first receives 50 minutes of direct instruction and then plays Cargo-Bot for 70 minutes. Interestingly, both groups experience the greatest learning gains directly after playing the video game. Thus, we observe that the game appears to have some educational benefit beyond simply contextualizing recursion.

Unfortunately, our attempt to repeat the experiment at a larger scale at a major university failed to produce statistically significant findings, and we conjecture that the approach was too unstructured. The game offers many puzzles of widely varying difficulty, and the unstructured approach does not tell students where they should focus their efforts, so it is unclear if students ever progress to the puzzles that involve more difficult recursive concepts such as passive flow.

In this paper, we hypothesize that a structured use of Cargo-Bot can improve students' ability to learn recursion. In particular, we propose a structure that (1) defines a set of learning goals with respect to recursion, (2) maps these learning goals to Cargo-Bot puzzles, (3) prescribes a minimal set of puzzles for students to solve, and (4) follows this gameplay with direct classroom instruction. We test this hypothesis by conducting an experiment with 204 students spread across two classes of CS2 at The University of Texas at Austin. One class serves as a control group: They receive four 50 minute lectures of direct instruction. The other class serves as the experimental group: They spend two class periods playing nine games of Cargo-Bot in a self-paced manner; they then receive two 50 minute class periods of direct instruction. We assess student learning by administering tests before and after the experiment.

We find with statistical significance that on calibrated assessments, the experimental group is better at writing recursive Java code than the control group ($p \leq 0.04612$). When we only consider the experimental students who complete the nine Cargo-Bot puzzles, the p-value falls below 0.0018.

This paper makes the following contributions:

- We define a set of learning goals related to recursive Java programming, and we map these learning goals to nine Cargo-Bot puzzles.

ICER'14, August 11-13, 2014, Glasgow, Scotland, UK.

http://dx.doi.org/10.1145/2632320.2632356.

- We design two condensed lectures that integrate examples from Cargo-Bot into the discussion and that replace an existing four lecture sequence.
- Using a study of 204 college students, we confirm our hypothesis that structured use of Cargo-Bot improves student learning of recursion.

2. RELATED WORK

Our work is informed by previous studies outlining best practices in teaching recursion [1, 6, 9–11, 18, 20, 23] and identifying common misconceptions of recursion. Students often create incorrect models of recursion [14, 29]. Most commonly they mistakenly model recursion as a loop structure and view recursion as iteration [6, 15, 24]. Moreover, students often struggle to understand the passive flow of recursion and the use of the stack for backtracking [22].

Our work is part of a growing effort to use visualizations and games to teach recursion [8, 25, 28]. For example, Alice is a 3D visual programming language that is useful for teaching basic recursion, but it is insufficient for teaching the more sophisticated details of recursion [4] because the state-less nature of Alice prevents the platform from visually representing recursion at a low level.

Two studies empirically study the use of games on student understanding of recursion. Chaffin, et al [2] use a game in which 16 students (including 14 upperclassmen) write depth-first search programs to visualize the traversal of a binary tree. Hulsizer [13] describes a similar experiment, reporting statistically significant results for a pool of 10 participants. Our work differs in two ways: We integrate gameplay with classroom instruction, and we use larger populations in our study.

As mentioned in the previous section, our work builds directly on that of Tessler et al. [26], who combine the unstructured use of Cargo-Bot and classroom instruction.

3. BACKGROUND: CARGO-BOT

Cargo-Bot is a game originally created for the Apple iPad by Two Lives Left[1]. Gameplay centers on a crane that moves and stacks a set of colored crates. Players write small visual programs to move the crates from an initial configuration to a goal configuration. The set of available instructions is quite small. The crane can be directed to (1) move left; (2) move right; or (3) move down and then up, in which case it attempts to pick up a crate if it is empty, or it drops its crate if it is not empty. Conditionals and procedure calls are also provided. Significantly, recursion is the only mechanism for repetition.

4. STRUCTURED LEARNING

To support structured learning, we first identify general subtopics within recursion, leveraging numerous resources such as existing course lecture materials, recursive problems on past course assessments, and online instructional tools such as CodingBat[2]. Our analysis reveals a few notable content clusters, and we give these clusters the following names to summarize their commonalities: recursion with conditionals, mutual recursion, recursive backtracking, and recursion with accumulators.

[1] `http://twolivesleft.com/CargoBot/`
[2] `http://www.codingbat.com/`

These clusters are mapped to the following learning goals:

1. Write recursive methods to progress toward some set of base cases.
2. Write code that conditionally selects one of multiple paths to make progress toward the base cases.
3. Trace a recursive call by visualizing the program stack.
4. Write code that uses passive flow to maintain state.
5. Utilize mutual recursion.
6. Utilize recursive calls as accumulators.

To scaffold student learning, we then align gameplay with instruction by mapping a set of Cargo-Bot puzzles to each learning goal. We find that Cargo-Bot puzzles bifurcate into two classes: (1) counting problems and (2) divide and conquer problems. This phenomenon is reflected in the initial branching in the tree of Cargo-Bot puzzles depicted in Figure 1. The tree is rooted with three basic puzzles that introduce students to the mechanics of Cargo-Bot and basic gameplay. From there, each of the two main branches moves through different recursive topics, roughly in order of increasing complexity, progressing from recursion using conditionals, to mutual recursion, to recursive backtracking, and finally to recursion with accumulators.

Puzzles in the left branch are solved by using the stack to count the number of times to execute a series of instructions. Puzzles in the right branch are solved using divide and conquer algorithms, focusing on conditioning recursion on the colors of crates. In our study, we prescribe a series of nine puzzles through the left, or counting, branch that strikes a balance between content coverage and brevity. These puzzles align with the learning goals #1–4 above and are shown connected by the red arcs (light grey arcs) in Figure 1. They are: Cargo-Bot 101, Transporter, Recurses, Go Left, Go Left 2, The Stacker, Clarify, Up the Greens, and Come Together.

4.1 Recursion in Cargo-Bot

Solutions to the easier puzzles utilize tail recursion, which is indistinguishable from `GOTO`-style control flow. Later puzzles require students to use recursive backtracking to maintain a counter that is implicitly stored as state on the program stack.

To illustrate this complexity, consider the puzzle "Come Together," the final puzzle in our set of prescribed puzzles. Figure 2 depicts the puzzle's goal state and its corresponding recursive solution. The execution of a Cargo-Bot program starts with the left-most instruction in the $F0$ function. From there, each subsequent instruction is executed left-to-right. We see that the $F1$ function is recursive: It calls itself whenever it encounters an empty column, and it effectively remembers the number of times that the crane has moved right, because the function moves left once for each time that it moves right.

4.2 Modifications to Cargo-Bot

Beyond identifying and establishing a prescribed course of Cargo-Bot content, we improve upon the game to support accompanying instruction and to improve students' experience with the game.

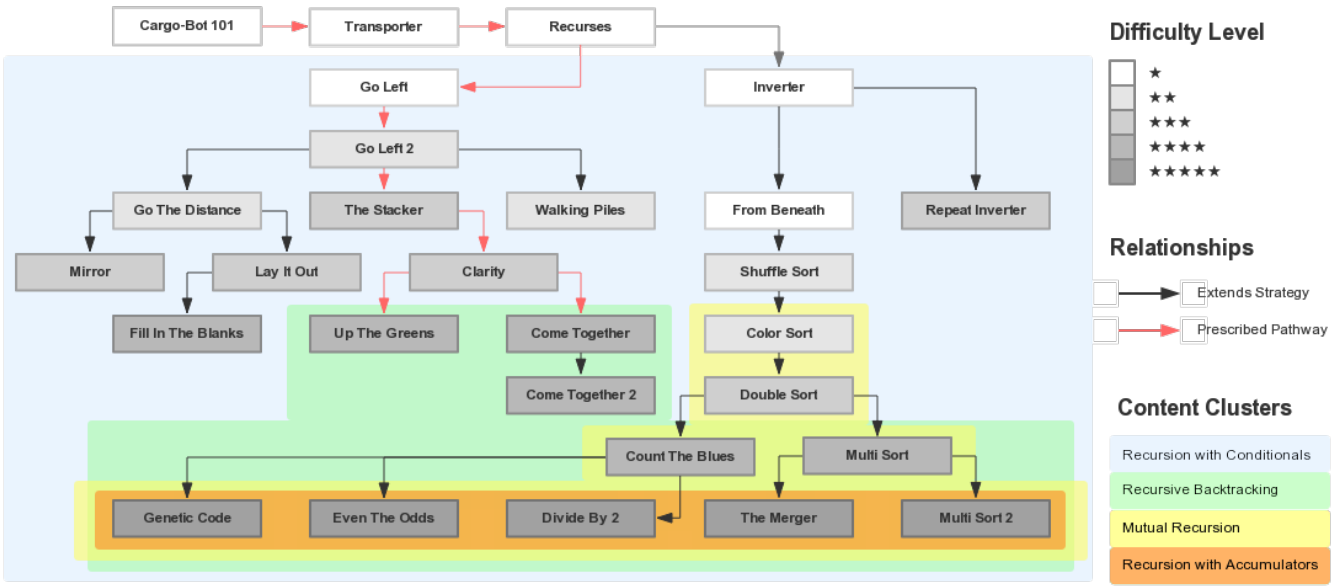

Figure 1: Taxonomy of Cargo-Bot puzzles.

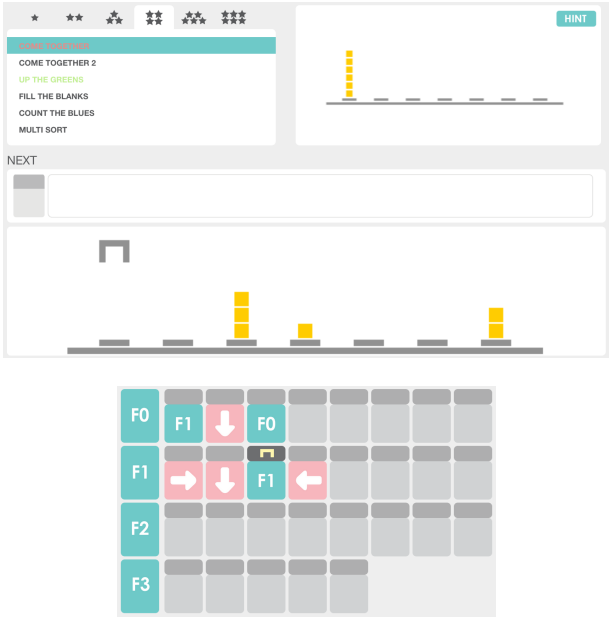

Figure 2: Cargo-Bot puzzle "Come Together".

4.2.1 *Visualizing the Stack*

Since an understanding of recursive solutions often requires an understanding of the program stack, we modify the game to include a visualization of the program stack, which is particularly helpful in counting and backtracking puzzles. These visual representations align with demonstrations of the program stack in the companion classroom lectures.

4.2.2 *Improving Game Feedback*

In the original game, players are rewarded with up to three stars depending on the length of their solution, but this metric often rewards complex, inelegant solutions, so we adjust the rating system to reward what we believe are the cleanest, most elegant solutions.

4.2.3 *Instrumentation*

To facilitate data collection, users enter a unique university ID when loading the initial page. All clickable actions are logged and tagged with the associated university ID and IP address. The log files can then be analyzed both at the individual level and in aggregate. Collected data include the number of attempts per puzzle, the ratings of correct solutions, the time spent on each puzzle, and the number of incorrect attempts at a solution.

5. EXPERIMENTAL DESIGN

Our experiment follows students in two sections of CS2 that are taught by the same instructor. One section of 136 students serves as the experimental group, while the other section of 187 students serves as the control group. Only those students consenting to the research study and completing all of the required parts of the experiment are included in our results, resulting in 88 students in the experimental group and 116 in the control group. These numbers are nearly equally proportional to their corresponding section total enrollments (65% *vs.* 62%, respectively), and there are no statistically significant differences in student population samples between the two groups.

5.1 Instruction and Gameplay

The existing CS2 course scope and sequence allots four 50 minute lectures to cover recursion: two on basic recursion and two on recursive backtracking. To compare our intervention to the *status quo*, the control group receives the four lectures as they are traditionally given, while students in the experimental group spend the first two lecture periods playing Cargo-Bot and the last two lecture periods receiving condensed lectures on recursion, one on basic recursion and the other on recursive backtracking. These two experimental lectures are given by one of the co-authors; for

the control group, the four lectures are given by the course instructor.

Students in the experimental group are given five days to complete the nine prescribed Cargo-Bot puzzles. They are allowed to play the game during class with guidance from the instructors and are encouraged to play on their own outside of class. During these first two lecture periods, the experimental group does not receive any formal instruction outside of the Cargo-Bot gameplay mechanics. The two condensed lectures cover the same content as the four lectures given to the control group, and they reference Cargo-Bot through worked examples. These lectures are condensed by omitting several worked examples of recursive Java code.

5.2 Evaluation

Each participating student takes a pre-test and a post-test, and each test evaluates student understanding of recursion by asking them to complete two tasks: (1) trace and explain the outcome of a recursive function and (2) write a recursive function to accomplish a given task. Each test also contains a survey that gauges student motivation and self-awareness: Students rate their abilities on a scale from "Strongly Agree" to "Strongly Disagree" regarding their understanding of recursion, their ability to follow the execution of a recursive function, and their ability to write a recursive function. The experimental group also rates their enjoyment of Cargo-Bot, rates their ability to play Cargo-Bot, and indicates whether they recognize that their Cargo-Bot solutions use recursion. All students take the pre-test prior to any instruction or formal discussion of recursion. Likewise, the post-test is given only after students complete the four 50 minute class meetings appropriate to their group.

5.2.1 Designing the Evaluations

To measure student understanding of recursion, the pre- and post-tests use performance tasks aligned with our descriptive taxonomy (see Figure 1). Each assessment is designed to be completed within the 10 minutes allotted for daily quizzes and to be structured similarly to previous daily quizzes. Our quizzes are constructed to reflect the following learning objectives:

- When **tracing** recursive functions, students will be able to (1) track function calls on the program stack and (2) explain the purpose of a given recursive function.
- When **writing** recursive functions, students will be able to use Java to (1) identify and construct appropriate base cases, (2) divide a problem into suitable subproblems, and (3) return values appropriately through both active and passive control flow.

Table 1 summarizes each of the tests' contents (Section 5.2.3 explains why there are four test forms). Figures 3 shows an example of a tracing problem that assesses students' ability to trace and understand recursive functions. In this example, an ideal answer to the second question is "the function counts the number of digits in the positive integer n".

Figures 4–7 show the writing problems for Forms A-D, respectively. For example, in Figure 7, the solution requires the programmer to maintain a counter, and an elegant recursive solution uses backtracking. Students must break the problem into subproblems, identify the base cases, and return the proper values up the stack.

	Form	**Tracing**	**Writing**
Pre-Test	A	Decimal-to-binary conversion	Greatest Common Denominator
	B	Count number of digits	Flood Fill
Post-Test	C	Modulus	Count subset sums
	D	Duplicate removal	Stairway

Table 1: Summary of test items. Forms A and B are pre-tests, while Forms C and D are post-tests.

```
public int foo(int n) {
    if (n == 0)
      return 0;
    return 1 + foo(n / 10);
}

What is the value of foo(12346)?

What do you think this function does?
```

Figure 3: A recursive tracing problem.

5.2.2 Grading Rubric

The grading rubric outlined in Table 2 is applied to all test forms to evaluate students' final test scores. The tracing problem is worth 6 points, and the writing problem is worth 14 points, for a total of 20 possible points.

Tracing Problem	**6**
Correct return value	3
Correct function description	3

Writing Problem	**14**
Non-recursive solution	0
Recursive solution, non-terminating	3
Recursive solution, terminating	6
Recursive solution, terminates w/progress	9
Recursive solution, correct, with:	14
— *Invalid Java syntax*	*-1*
— *Unnecessary base cases*	*-1*
— *Unnecessary function calls*	*-1*
— *Extra return values*	*-1*

Table 2: Grading Rubric for pre- and post-tests.

5.2.3 Test Validation

To strengthen the reliability of our assessments, each of the pre- and post-tests has two versions: Students in both groups randomly take either Form A or Form B as a pre-test and either Form C or Form D as a post-test. Because each test is open-ended, we assume that the tests may be uneven in difficulty. To accurately evaluate students across all combinations of test forms, we take a two-pronged approach:

1. Each of the open-ended responses is graded by the same grader using a common rubric (see Table 2).

```
   Given two numbers, write a recursive
   function that returns their greatest
   common divisor (the largest number
   that is a factor of both numbers).

Examples:
   gcd(12, 36) returns 12
   gcd(14, 10) returns 2

Please implement your method in proper Java
syntax and use the following method signature:

public int gcd(int a, int b)
{
  // ...
}
```

Figure 4: The recursive writing problem for Form A.

2. We create a baseline set of scores by asking 36 current computer science majors who have already completed CS2 to take one or two of the four tests. Their raw scores are used to provide summary statistics for each test form, which allow us to establish a mapping from raw scores to standard z-scores. The summary statistics of the baseline group are shown in Table 3.

	Tracing		**Writing**		**Total**	
	mean	*stdev*	*mean*	*stdev*	*mean*	*stdev*
Form A	5.12	1.76	7.11	6.92	12.11	6.88
Form B	4.58	2.32	11.14	3.98	15.43	4.59
Form C	5.81	0.75	7.75	5.14	13.56	5.38
Form D	4.06	2.84	10.44	4.95	14.50	6.48

Table 3: Baseline Scores from Upperclassmen. Forms A and B are pre-tests, while Forms C and D are post-tests.

5.3 Student Diversity

To minimize bias and priming effects on survey and performance data, the gender and race/ethnicity data associated with each ID are retrieved from institutional data records only after the student assessments are collected (see Table 4). Here, the ethnic label "Hispanic" supersedes the race label "White"; university institutional data equate the label "White" with "Non-Hispanic White". The "Other" category encapsulates students who identify as races other than those listed, as multiracial, or who choose to withhold this information from public records.

6. RESULTS

By examining the performance gains from the pre-test to post-test scores, we show by a statistically significant margin that students who play our prescribed pathway through Cargo-Bot score higher than those who do not.

6.1 Statistical Tools

To account for variation in the two different forms of the pre- and post-tests, we calculate the standard z-score for each student. A z-score gives the relationship of a given raw score to the mean score and standard deviation of the

```
   In any modern image editor, you have
   access to the "fill" or paint bucket tool,
   which fills all instances of a target
   color with a given replacement color
   at some given location in the image.
   Your task is to implement this algorithm
   recursively.

Example:
   If fill is called on a pixel in the middle
   area in the image on the left below,
   fill should return the image on the right.

   Assume you have access to the Image
   class with the following methods:

     Color getColor(int x, int y) -
     returns the color at (x,y) or null if
     the given (x,y) is outside of the image.

     void setColor(int x, int y, Color c) -
     sets the color at (x,y) or throws an
     error if (x,y) is outside of the image.

   Also, you can easily compare two colors
   using the equals method, e.g.,

     color1.equals(color2).

Please implement your method in proper Java
syntax and use the following method signature:

public void fill(Image img, Color targetColor,
            Color replaceColor, int x, int y)
{
  // ...
}
```

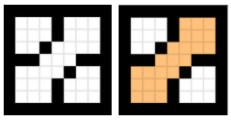

Figure 5: The recursive writing problem for Form B.

population as a whole,

$$z = \frac{score - \mu}{\sigma}$$

Here, the population as a whole is defined by the baseline scores, as explained in Section 5.2.3.

By using z-scores instead of raw test scores, we account for the variance in difficulty among tests when comparing the pre-test scores to the post-test scores. Negative z-scores in our study are to be expected, since the baseline represents students who have completed the course. The difference between the post-test and pre-test z-scores measures a student's performance gains. A positive difference indicates that a student has improved over time, while a negative difference means that they have regressed over time.

We use a one-way Type-III ANCOVA (Analysis of Covariance) model to minimize the variance in error of our results. We use four possible combinations of pre-tests and post-tests in our experiment. By randomly assigning each student a pre-test and a post-test, we mitigate the potential for systemic bias among particular test forms. We must, however, account for potential sources of variance, such as the pre-test scores, that could affect the outcome of the study. Following

	Female	Male	Asian	Black	Hispanic	White	Other
Control	21.3%	78.7%	29.31%	2.59%	28.45%	37.07%	2.59%
Experimental	22.3%	77.7%	29.55%	4.55%	17.05%	46.59%	2.27%
Total	**21.7%**	**78.3%**	**29.41%**	**3.43%**	**23.53%**	**41.18%**	**2.45%**

Table 4: Demographic breakdown of student groups.

```
    Given an array of integers, arr, and a
    target sum, sum, write a recursive
    function that returns the number of
    subsets in arr that contain elements
    that add up to sum.

Example:
    given arr = [1,2,3,2] and sum = 4, there
    are 2 subsets, [1,3] and [2,2] that sum
    to 4, so countSubsets(arr, sum) returns 2.

Please implement your method in proper Java
syntax and use the following method signature:

public int countSubsets(int[] arr, int sum) {
{
  // ...
}
```

Figure 6: The recursive writing problem for Form C.

```
    A child can climb stairs 1, 2, or 3
    steps at a time. Write a recursive
    function that returns the number
    of distinct ways the child can
    climb n stairs.

Please implement your method in proper Java
syntax and use the following method signature:

public int stairways(int n)
{
  // ...
}
```

Figure 7: The recursive writing problem for Form D.

the practices established by Dimitrov and Rumrill, Jr. [5] to evaluate improvement between pre-tests and post-tests, we use the ANCOVA model with the pre-test scores serving as a covariate. This model allows us to measure the post-test score as it relates to a given independent variable such as group (control or experimental) or gender, using the pre-test as a covariate. To extend our analysis, we use a Tukey test to determine the differences among categories, such as gender or group.

6.2 Analysis of Student Performance Gains

Figure 8 shows the increase from the average pre-test z-score to average post-test z-score for the control and experimental groups. While the experimental group clearly improves more than the control group, we find that the increase in scores is not statistically significant. When we run the ANCOVA model for each group with the pre-test as a covariate, we see statistically significant results for the writing score ($p \leq 0.04612$) but not for the tracing or total score.

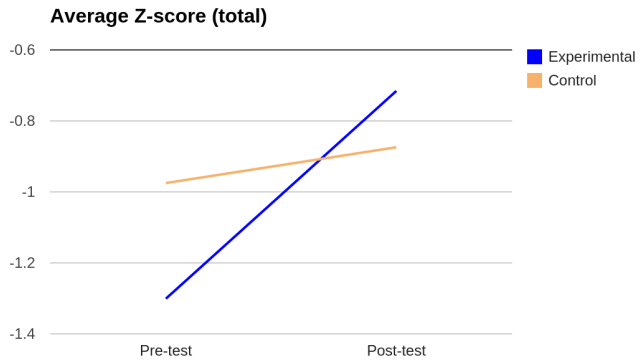

Figure 8: Improvement in terms of z-scores from the pre-test to the post-test.

The results of the tracing scores are consistent with those of Tessler et al.'s experiment [26]. When we use ANCOVA to model gains by gender and race/ethnicity with pre-test as a covariate, we find that gender and race/ethnicity are not significant factors for gains in total, tracing, or writing scores across the entire group.

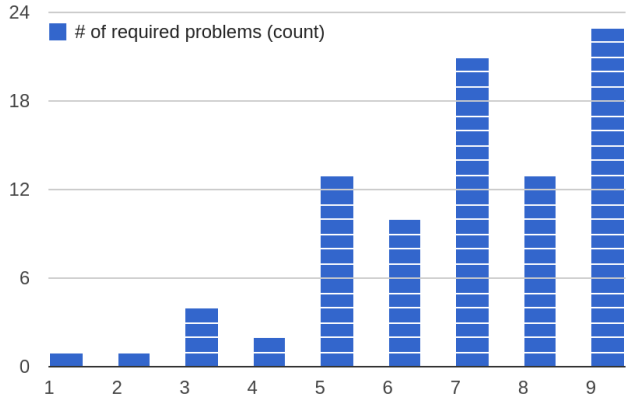

Figure 9: Number of puzzles completed by students in the experimental group.

Figure 9 shows the number of prescribed Cargo-Bot puzzles completed by the students in the experimental group. Only 26.14% percent of students completed all nine assigned puzzles. 40.9% completed at least eight and 64.77% completed at least seven of the prescribed puzzles. On average, students completed seven of the nine required puzzles and a total of eight puzzles on average. We see in Table 5 that those students who complete eight or nine puzzles score higher than the control group by a statistically significant margin. By contrast, students who complete seven or fewer

Exp *vs* Control	Total Score (p)	Writing Score (p)
6 puzzles	0.8701	0.8391
7 puzzles	0.8706	0.9812
8 puzzles	0.0439	0.0746
9 puzzles	0.0043	0.0018

Table 5: Results of experimental group performance against the control sorted by number of puzzles completed. The completion of eight or nine puzzles correlates to a statistically significant difference in group performance ($\alpha < 0.05$).

puzzles do not score significantly higher than the control group.

Finally, we explore the effect of our intervention on final exam performance, which takes place after students have submitted a significant recursive programming assignment. We find that the experimental group retains some of their advantage on the final exam's recursion-related questions, which are created and graded by the course's instructional staff. Those who complete at least 8 prescribed puzzles perform better than the control group in both tracing ($p \leq 0.03997$) and writing ($p \leq 0.08867$), though the latter is not statistically significant. The advantage is also not statistically significant for those who complete just 7 of the puzzles ($p \leq 0.3851$ for tracing, $p \leq 0.3469$ for writing).

6.3 Survey Results

Our assessments ask students to rate their attitudes and abilities in relation to recursion. Students are shown the following three statements and are asked to choose "Strongly Agree", "Agree", "Neither Agree nor Disagree", "Disagree", or "Strongly Disagree" in response:

- I understand recursion.
- I can follow the execution of a recursive function.
- I can write a recursive function.

In comparing the experimental and control groups, we find no statistically significant results from the surveys. We do, however, see a few interesting points. We find that student attitudes do not always match student performance. We see that most students enjoy playing Cargo-Bot, but less than 36% of the students are confident in their Cargo-Bot abilities, while over 62% of these same students are confident in their ability to write a recursive function (see Figure 10).

7. CONCLUSIONS

In this paper, we have proposed a new method of teaching recursion that uses Cargo-Bot in conjunction with classroom instruction. Our results are encouraging both because of the strong statistical evidence of the approach's effectiveness and because of the fairly large number of students—204—involved.

While these results are encouraging, empirical educational studies are clearly methodologically difficult, so we plan to conduct additional studies that refine the methodology and that explore other related questions. For example, direct instruction might be considered the most passive of educational methods, so it would be interesting to compare self-guided game playing with self-guided problem solving using instruments similar to CodingBat.

Acknowledgments.

We thank Mike Scott, Lara Schmidt, and Zhao Song for their help in conducting our experiment on their class. We thank the students who participated in our study and Alex Suchman for his guidance on statistical analysis. Our work is partially supported by the National Science Foundation under grant #CNS-1138506 and by OnRamps coordinated by The University of Texas at Austin. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of its funding sources.

References

[1] Alan C. Benander and Barbara 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.

[2] Amanda Chaffin, Katelyn Doran, Drew Hicks, and Tiffany 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, New York, NY, USA, 2009. ACM.

[3] Diana I. Cordova and Mark R. Lepper. Intrinsic motivation and the process of learning: Beneficial effects of contextualization, personalization, and choice. *Journal of Educational Psychology*, 88(4):715–730, 1996.

[4] Wanda Dann, Stephen Cooper, and Randy Pausch. Using visualization to teach novices recursion. In *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '01, pages 109–112, New York, NY, USA, 2001. ACM.

[5] Dimiter M Dimitrov and Phillip D Rumrill, Jr. Pretest-posttest designs and measurement of change. *Work: A Journal of Prevention, Assessment and Rehabilitation*, 20(2):159–165, 2003.

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

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

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

[9] David Ginat and Eyal Shifroni. Teaching recursion in a procedural environment—how much should we emphasize the computing model? *SIGCSE Bulletin*, 31(1):127–131, March 1999.

[10] James Eugene Greer. *An empirical comparison of techniques for teaching recursion in introductory computer sciences*. Ph.D. dissertation, The University of Texas at Austin, May 1987.

[11] Katherine Gunion, Todd Milford, and Ulrike Stege. Curing recursion aversion. *SIGCSE Bulletin*, 41(3):124–128, July 2009.

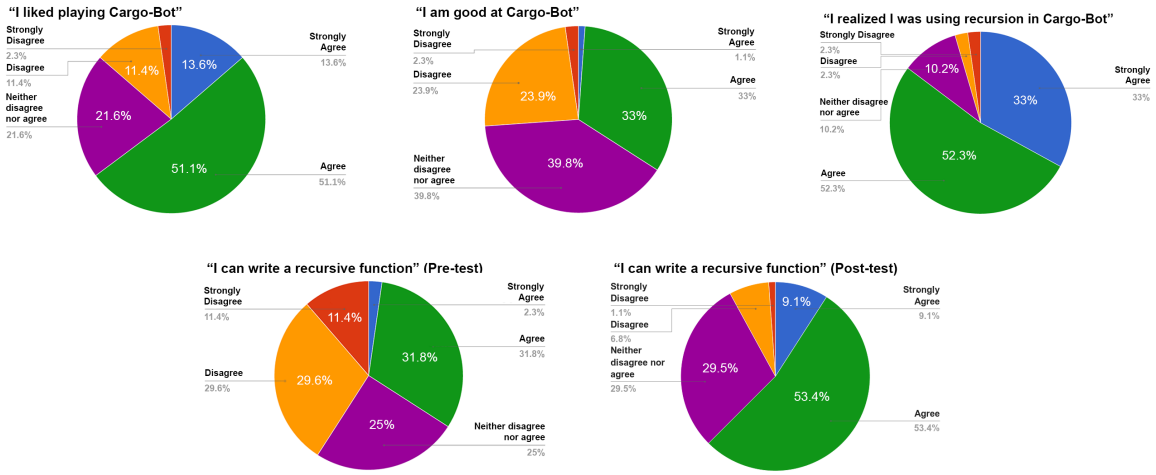

Figure 10: Post-test (except where noted) survey results for experimental group.

[12] WenJung Hsin. Teaching recursion using recursion graphs. *Journal of Computing Sciences in Colleges*, 23(4):217–222, April 2008.

[13] Andrew Hulsizer. *Teaching Recursion Through Interactive Media*. Masters thesis, The University of Texas at Austin, 2011.

[14] Hank Kahney. What do novice programmers know about recursion? In Elliot Soloway and James C. Spohrer, editors, *Studying the Novice Programmer*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1989.

[15] Claudius M. Kessler and John R. Anderson. Learning control flow: Recursive and iterative procedures. In Elliot Soloway and James C. Spohrer, editors, *Studying the Novice Programmer*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1989.

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

[17] Dalit Levy and Tami Lapidot. Recursively speaking: analyzing students' discourse of recursive phenomena. *SIGCSE Bulletin*, 32(1):315–319, March 2000.

[18] Peter L. Pirolli and John R. Anderson. The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology*, 39(2):240–272, June 1985.

[19] Irene Polycarpou, Ana Pasztor, and Malek Adjouadi. A conceptual approach to teaching induction for computer science. *SIGCSE Bulletin*, 40(1):9–13, March 2008.

[20] Anthony Robins, Nathan Rountree, and Janet Rountree. My program is correct but it doesn't run: a review of novice programming and a study of an introductory programming paper. *Technical Report*, OUCS-2001-06, 2001. University of Otago.

[21] Manuel Rubio-Sánchez and Isidoro Hernán-Losada. Exploring recursion with Fibonacci numbers. *SIGCSE Bulletin*, 39(3):359–359, June 2007.

[22] Tamarisk Lurlyn Scholtz and Ian 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, New York, NY, USA, 2010. ACM.

[23] Amber Settle. What's motivation got to do with it? A survey of recursion in the computing education literature. *Technical Reports*, Paper 23, 2014. DePaul University. `http://via.library.depaul.edu/tr/23`.

[24] Raja Sooriamurthi. Problems in comprehending recursion and suggested solutions. *SIGCSE Bulletin*, 33(3):25–28, June 2001.

[25] John Stasko, Albert Badre, and Clayton Lewis. Do algorithm animations assist learning?: an empirical study and analysis. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, CHI '93, pages 61–66, New York, NY, USA, 1993. ACM.

[26] Joe Tessler, Bradley Beth, and Calvin 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, New York, NY, USA, August 2013. ACM.

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

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

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