AN INTERVENTION STRATEGY TO HONE STUDENTS' CODE UNDERSTANDING SKILLS

Benito Mendoza
New York City College of Technology | CUNY
300 Jay Street | Brooklyn, NY 11201
718.260.5885 | bmendoza@citytech.cuny.edu

Laura Zavala
Medgar Evers College | CUNY
1650 Bedford Ave. | Brooklyn, NY 11225
718.270.6128 | rzgutierrez@mec.cuny.edu

ABSTRACT

The main focus in programming courses is usually on writing code; students are asked to write programs after only being taught the syntax rules and a few examples. We believe that more emphasis needs to be put in code understanding skills, which are precursor skills to writing code. In this paper, we present the results of an intervention strategy we implemented with the goal of honing students' program understanding skills. Automatic Item Generation was used to generate a set of practice exercises individualized to each student's needs. Results of pre and post-assessments from the intervention group and a control group show the benefits of dedicating time to program understanding.

INTRODUCTION

Although the need for considerable practice in introductory computer programming courses is indisputable and widely acknowledged, asking novice students to write programs after only teaching them syntax rules (of a programming language) and a few examples, might not be the right approach. High failure rates abide in CS1 courses all over the world with a pass rate estimated to be around 67.7% [1]. The emphasis in programming courses is mainly in writing code. This might be adequate for advanced, upper-level courses. However, for introductory courses, the fact that code understanding skills are a precursor to code writing skills should not be ignored [2, 3].

Pedagogical research has shown that instruction that relies more heavily on the study of worked examples is more effective and efficient for learning and transfer than instruction consisting of problem-solving [4, 5, 6]; problem-solving requires an enormous amount of cognitive processing capacity that interferes with learning to understand [4]. This underlying idea has been explored in several problem solving oriented disciplines. For example, Sweller and Cooper [7] compared the efficacy of a worked-examples approach versus a problem-solving approach. In their experiments, some students learnt certain algebraic processes by solving problems, while other students learnt the same algebraic processes by studying complete solutions to those problems (i.e., worked examples). They found that students who learned algebra with worked-examples performed significantly better than those who were asked to solve problems instead. In the CS domain, practicioners have addressed the need for a focus on code understanding before program writing in introductory programming courses [1, 2, 3, 8, 9,10].

Zavala and Mendoza [2, 3] argue that students should be able to not only read code before they can write code, but also manipulate code that is given to them (i.e., modify or use). Formally, they state that there are three phases that students should sequentially master in the process of learning to write programs: code comprehension, code

manipulation, and code writing. In addition to these competencies, there are other skills, called companion skills, which should also be stressed in all three phases, many of which are usually under-looked in introductory programming courses. Namely, identifying programming constructs (variables, data types, expressions, function definitions, function calls, parameters, etc.), explaining code (being able to verbally explain what a piece of code does), understanding technical documentation (i.e., APIs), and refactoring existing code.

Although code comprehension and manipulation work might be covered in programming courses through sample solutions, design roadmaps, implementation hints, and starter code, the emphasis is, by far, on code writing. Providing students with all these resources does not really constitute practice on code comprehension and manipulation; it is not the main focus and students are rarely or never assessed on these competencies [3]. In this paper, we present the results of an intervention strategy we implemented with the goal of honing students' code understanding skills. For a period of two weeks, students in an introductory programming course were given practice that was focused on code comprehension and consisted mainly on reading, tracing, and making small changes to existing programs. Automatic Item Generation (AIG) was used to generate a set of practice exercises individualized to each student's needs. We present the results of pre and post-assessments from the intervention group and a control group. Our results show the benefits of dedicating time to code understanding.

BACKGROUND

Several works have demonstrated that instruction focused on worked-examples is more effective for learning than instruction consisting of problem-solving. Kirschner, Sweller, and Clark [11] recommend providing students with worked-examples and process worksheets, which should provide a description of the steps they should go through when solving the problem as well as hints or rules of thumb that may help to successfully complete each step. In a similar effort, Van Gog [12] studies the effects of worked-examples compared to problem-solving and shows that it is not strictly necessary to alternate example study and problem solving but that example study only and example study alternated with problem-solving is more effective and efficient than problem-solving only.

Venables et al. [13] and Lopez at al. [14] design and analyzed an exam that students took at the end of an introductory course on programming. Both found that the ability of students to explain and trace code correlated positively with their ability to write code. Venables et al. concluded that when students are reasonably capable of both tracing and explaining, the ability to systematically write code emerges. Porter and Simon [15] evaluated the competency of students that have completed their first one or two courses in computer science and concluded that many students do not know how to program at the conclusion of their introductory courses. Lister et al. [16] evaluated students from seven countries at the conclusion of their CS1 course on two abilities: (i) predicting the outcome of a program, and (ii) completing a near-complete program. They noted that poor performance in one or both abilities might be an explanation for the fact that many students do not know how to program at the conclusion of their introductory courses. Similar studies in [2, 3] further tested students on both code comprehension skills as well as code writing skills of similar complexity. They showed that weaknesses in understanding and manipulating existing code correlate with students' abilities to write code.

PROBLEM FORMULATION

The high-level goal for our study was to assess the impact of explicitly spending time on code understanding in a CS0 course. Specifically, we seek to answer the following question: Do students who explicitly spend time on code reading and manipulation activities progress at a faster pace when learning to write code than students who do not?

What is the output of the following program?

```
for number in range(1, 13):
    if (number % 2) == 0 and (number % 4) ==0:
        print('Multiple of 2 and 4')
    elif (number % 2) == 0:
        print('Multiple of 2 only')
    elif (number % 4) == 0:
        print('Multiple of 4 only')
    else:
        print('None')
```

(a)

Write a Python program for each of the following lists. The program should print what is indicated on the Output column, using the list in the List column. You will write a total of 3 programs. The first one is given for you as example.

List	Output
fruits = ['apples', 'oranges', 'pears', 'apricots', 'grapes']	All the fruits that start with 'a'
cities = ['Rome', 'London', 'Paris', 'Berlin', 'Madrid', 'Lisbon']	All the cities that end with 'n'
colors = ['red', 'green', 'blue', 'yellow', 'brown', 'cyan']	All the colors that start with 'b'
animals = ['bear', 'python', 'peacock', 'kangaroo', 'platypus']	All the animals that start with 'p'

Example:

```
fruits = ['apples', 'oranges', 'pears', 'apricots', 'grapes']
for fruit in fruits:
    if fruit[0] == 'a':
        print(fruit)
```

(b)

```
apples
apricots
```

Write a function that has a string parameter, w. If w starts with 'f' the function prints 'fizz'. If word ends with 'z', the function prints 'buzz'. If both conditions are true, the function prints 'fizzbuzz'. Otherwise, the function prints the original value of w.

Examples of what the function should print for different values of w:

w	output	w	output
friend	fizz	**fizz**	fizzbuzz
quiz	buzz	**futz**	fizzbuzz
nice	nice	**puzzle**	puzzle

```
def fizzbuzz(w):
```

(c)

FIGURE 1. SAMPLE QUESTIONS GIVEN IN THE PRE-ASSESMENT: (A) CODE READING; (B) CODE MANIPULATION; (C) CODE WRITING.

METHODOLOGY

Two sections of an introductory programming course (CS0 course) were used for the study. The course is taught at a four-year urban college and uses Python programming language. One section of the course was used to test an intervention strategy implemented with the goal of honing students' code understanding skills. The other section was used as a control group. Students in both groups were given a pre-test to evaluate their code comprehension, code manipulation, and code writing competencies. The pre-test was given past mid-semester, when students had already learned some basic programming. Figure 1 shows some of the questions given in the pre-assessment.

Automated Item Generation

AIG is an approach for developing test-items or questions for exams, automatically by a program [18]. The most common AIG approach is based on the use of test-item templates with embedded variables and formulas. The variables and formulas in the template are resolved by a computer program with actual values to generate test-items. Thus, hundreds or even thousands of test-items can be generated with a single test-item template, as exemplified in Figure 2. The obvious advantage of an AIG system is its ability to produce high volumes of test-items and therefore numerous different tests with the same difficulty and quality. Current approaches to AIG vary by the method used for giving values to the variables: a text [19], mathematical equations [18, 20], and more.

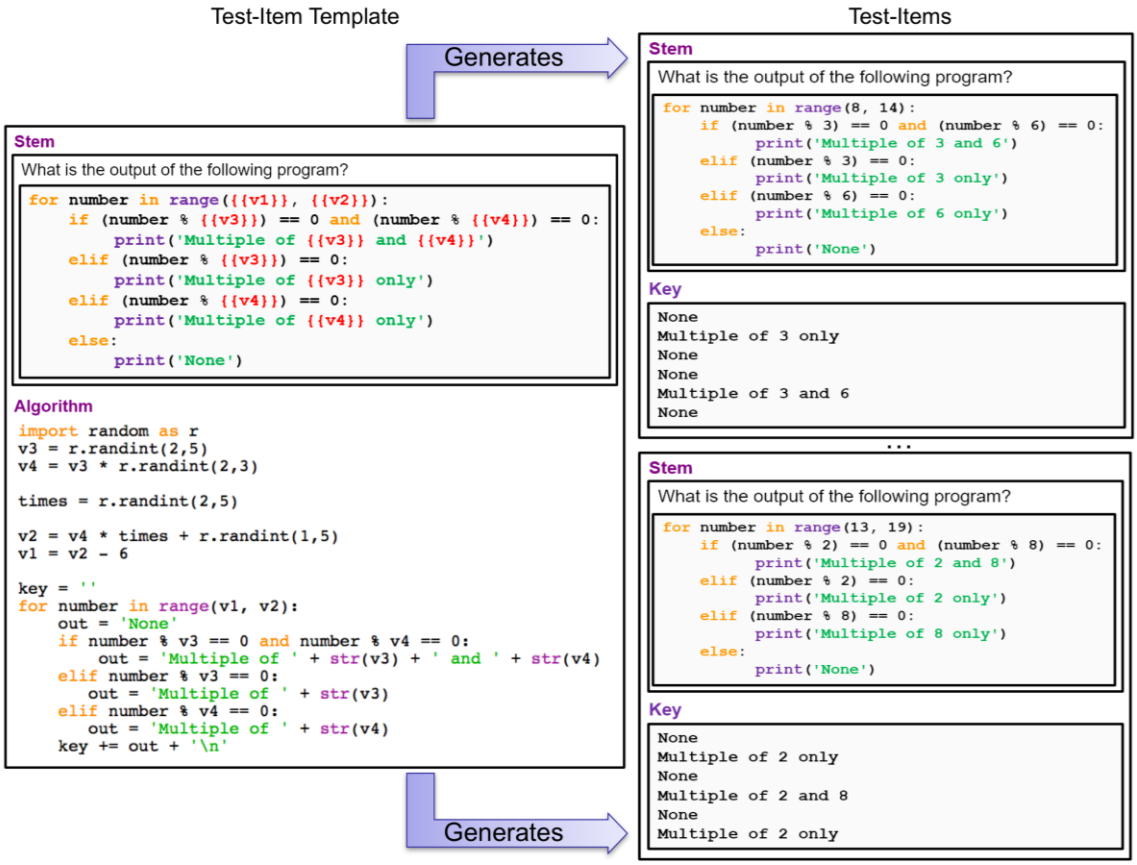

FIGURE 2. A CONCEPTUAL EXAMPLE OF AIG: IT SHOWS A TEST-ITEM TEMPLATE, THE ALGORITHM TO INSTANTIATE IT, AND TWO OF THE HUNDREDS OF QUESTIONS THAT CAN BE GENERATED WITH THIS TEMPLATE.

AIG was used to generate a set of practice exercises individualized to each student's needs in the intervention group. We incorporated basic AIG functionality into an existing self-assessment and practice tool for students learning computer programming [17]. The exercises were focused on code comprehension and consisted mainly on reading, tracing, and making small changes to existing programs. Students worked on their exercises for a period of two weeks. Some exercises were given to students as in-class lab assignments and others as homework assignments. Nothing else was covered during that period of time in the intervention group while the control group continued with the schedule as planned. The schedule was re-taken for the intervention group after the time dedicated to implementing the intervention strategy. A post-assessment was given to students in both groups near the end of the semester to evaluate them on the same skills than in the pre-test (with some overlap in content). At that point in the semester students are expected to be writing code. We were interested in observing whether there was any significant difference in the progress made in learning to code between the students from the two groups.

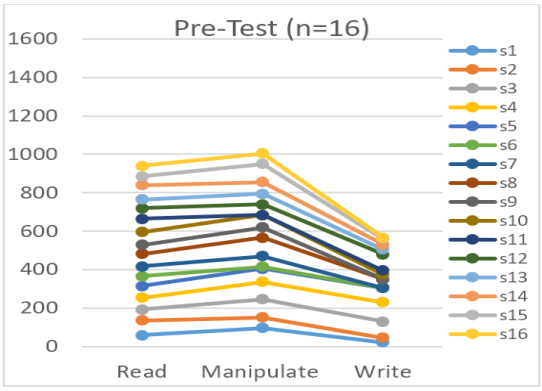

FIGURE 3. PRE-TEST RESULTS: CONTROL GROUP CUMULATIVE SCORES BY LEVEL

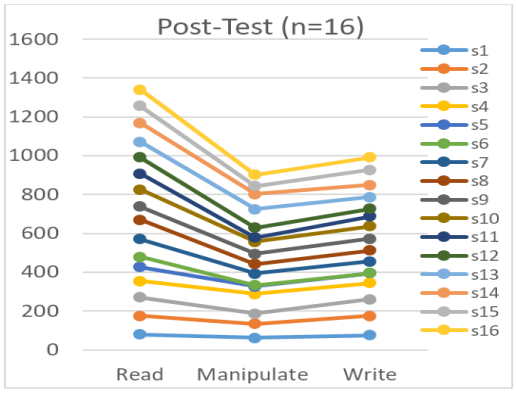

FIGURE 4. POST-TEST RESULTS: CONTROL GROUP CUMULATIVE SCORES BY LEVEL

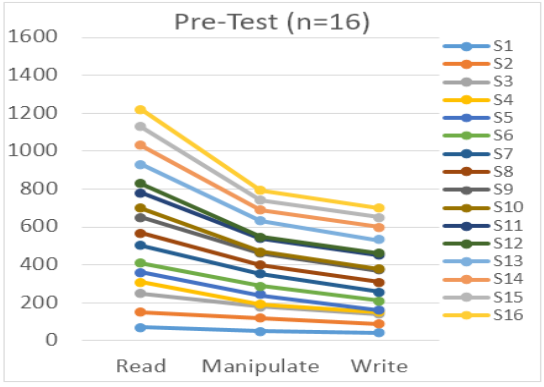

FIGURE 5. PRE-TEST RESULTS: INTERVENTION GROUP CUMULATIVE SCORES BY LEVEL

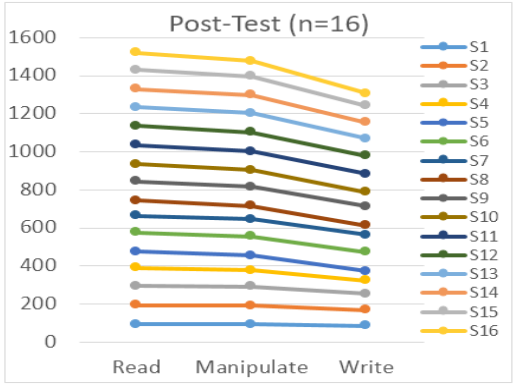

FIGURE 6. POST-TEST RESULTS: INTERVENTION GROUP CUMULATIVE SCORES BY LEVEL

RESULTS

Figures 3 to 6 show the results obtained in the pre and post assessments from the control group and the intervention group, respectively. The stacked line charts depict the cumulative students' scores by skill level: 1) read, 2) manipulate, and 3) write. Students have been anonymized (s1, s2, etc.). The evaluation instrument (quiz) for each skill level had a max score of 100. For most of the students code reading is the most dominant skill,

followed by code manipulation, while code writing is the weakest one, which is in line with previous studies [2, 3, 13, 16].

Both groups exhibit an improvement on all the skills towards the end of the course. However, students who explicitly spend time on code reading and manipulation exercises show a greater improvement than students who do not. All the students in the intervention group finished at a proficient level in code reading (Figure 6), while some in the control group are not (Figure 4). A larger difference in results is observed for code manipulation and code writing. Students in the intervention group greatly improved in such skills, and can be considered proficient in both as well. In fact, as can be seen on figure 6, they are almost at the same level on all skills. Although students in the control group also exhibit an improvement, the improvement is not that much for code manipulation and is minimal for code writing. Our results support the pedagogical research theories [4, 5, 6] that suggest relying more heavily on the study of worked-examples than on instruction consisting of problem solving. In our case we alternate example study with problem-solving, which as stated in [12] is more effective and efficient than problem-solving only.

CONCLUSIONS AND FUTURE WORK

We evaluated an intervention strategy implemented with the goal of honing students' code understanding skills. AIG was used to generate a set of practice exercises individualized to each student's needs. Results obtained from the intervention group and a control group show the benefits of dedicating time to code understanding. Students that participated in the intervention strategy show a greater improvement in learning to code over the students in the control group.

Future work will focus on more comprehensive approaches to address code understanding skills in computer programming courses. We are interested in the tantamount to the worked-examples approaches discussed earlier [4, 5, 6, 7]. Further, we believe emphasis in code understanding skills is something that should continue all the way through high-level courses, not only in introductory ones.

AIG is a convenient and helpful approach to automatically generate a large number of practice items, different for each student, but with the same difficulty and quality. By relieving the teacher from the burden of having to generate several practice questions for each student, it greatly facilitates an approach like the one presented in this paper. We are also working on AIG by investigating semantic, AI-based approaches that will allow a greater flexibility in the type of items that can be generated.

REFERENCES

[1] McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagen, D., Kolikant, Y., Laxer, C., Thomas, L., Utting, I., and Wilusz,T., A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First- year CS Students, *SIGCSE Bulletin,* 33, no. 4, 125- 140, 2001.

[2] Zavala, L., Read, Manipulate, and Write: A study of the role of these cumulative skills in learning computer programming, Proceedings of the ASEE NE 2016 Conference, April 28-30, University of Rhode Island, Kingston, RI, 2016.

[3] Zavala, L., and Mendoza, B., Precursor skills to writing code. *Journal of Computing Science in Colleges,* 32, 3, 149-156, 2017.

[4] Sweller, J., Cognitive Load During Problem Solving: Effects on Learning. Cognitive Science, 12: 257–285, 1988.
[5] Atkinson, R. K., Derry, S. J., Renkl, A., and Wortham, D., Learning from examples: Instructional principles from the worked examples research. Review of Educational Research, 70, 181-214, 2000.
[6] Sweller, J., Van Merriënboer, J. J. G., and Paas, F. G. W. C., Cognitive architecture and instructional design. Educational Psychology Review, 10, 251-295, 1988.
[7] Sweller, J., and Cooper, G. A., The use of worked examples as a substitute for problem-solving in learning algebra, *Cognition and Instruction,* 2, no. 1, 59–89, 1985.
[8] Guzdial, M., and Robertson, J., Too much programming too soon? *Communications of the ACM*, 53, no. 3 10-11, ACM, March 2010.
[9] Guzdial, M., What's the best way to teach computer science to beginners? *Communications of the ACM*, 58, no. 2,12-13, ACM, January 2015.
[10] Linn, M. C., and Clancy, M. J., The case for case studies of programming problems, *Communications of the ACM 35*, no. 3, 121-132, ACM, March 1992.
[11] Kirschner, P.A., Sweller, J., and Clark, R.E., Why minimal guidance during instruction does not work: an analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching, *Educational Psychologist,* 41, no. 2, 75-86, 2006.
[12] Van Gog, T., Kester, L., and Paas, F., Effects of worked examples, example-problem, and problem-example pairs on novices' learning, *Contemporary Educational Psychology,* 36(3), July 2011.
[13] Venables, A., Tan, G., and Lister, R., A closer look at tracing, explaining and code writing skills in the novice programmer. Proceedings of the Fifth International Conference on Computing Education Research (ICER '09), ACM, New York, NY, USA, 117-128, 2009.
[14] Lopez, M., Whalley, J., Robbins, P., and Lister, Relationships between reading, tracing and writing skills in introductory programming. 4th International Workshop on Computing Education Research, Sydney, Australia, 101–112, 2008.
[15] Porter, L., and Simon, B., Retaining nearly one-third more majors with a trio of instructional best practices in CS1, *Proceedings of the ACM Computer Science Education (SIGCSE '13). 44th Annual Technical Symposium*, 165-170, ACM, 2013.
[16] Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., and Thomas, L., A multi-national study of reading and tracing skills in novice programmers, *Working group reports from ITiCSE on Innovation and Technology in Computer Science Education*, 119-150, ACM, 2004.
[17] Mendoza, B., Reyes-Alamo, J., Wu, H., Carranza, and A., Zavala, L., iPractice: a self-assessment tool for students learning computer programming in an urban campus. *Journal of Computing Science in Colleges,* 31, 3, 93-100, 2016.
[18] Gierl, M. J., and Lai, H, The Role of Item Models in Automatic Item Generation. *International Journal of Testing*, 12(3), 273-298, 2012.
[19] Karamanis, N., Ha, L. A., and Mitkov, R., Generating multiple-choice test items from medical text: A pilot study. *Paper presented at the Fourth International Conference Natural Language Generation Sydney*, Australia, 2006.
[20] Liu, B., Chen, H., and He, W., A framework of deriving adaptive feedback from educational ontologies, *Proceedings of the 9th International Conference for Young Computer Scientists*, Zhang Jia Jie Hunan, China, 2008.