An Analysis of Incorporating Small Coding Exercises as Homework in Introductory Programming Courses

Dr. Alex Daniel Edgcomb, zyBooks

Alex Edgcomb finished his PhD in computer science at UC Riverside in 2014. Alex works with zyBooks.com, a startup that develops interactive, web-native textbooks in STEM. Alex also works as a research specialist at UC Riverside, studying the efficacy of web-native content for STEM education.

Prof. Frank Vahid, University of California, Riverside

Frank Vahid is a Professor of Computer Science and Engineering at the Univ. of California, Riverside. His research interests include embedded systems design, and engineering education. He is a co-founder of zyBooks.com.

Prof. Roman Lysecky, University of Arizona

Roman Lysecky is an Associate Professor of Electrical and Computer Engineering at the University of Arizona. He received his Ph.D. in Computer Science from the University of California, Riverside in 2005. His research interests include embedded systems, runtime optimization, non-intrusive system observation methods, data-adaptable systems, and embedded system security. He has recently coauthored multiple textbooks, published by zyBooks, that utilize a web-native, interactive, and animated approach, which has shown notable increases in student learning and course grades.

Dr. Susan Lysecky, zyBooks

Susan received her PhD in Computer Science from the University of California, Riverside in 2006. She served as a faculty member at the University of Arizona from 2006-2014. She has a background in design automation and optimization for embedded systems, as well as experience in the development of accessible engineering curricula and learning technologies. She is currently a Senior Content Developer at zyBooks, a startup that develops highly-interactive, web-native textbooks for a variety of STEM disciplines.

An Analysis of Incorporating Small Coding Exercises as Homework in Introductory Programming Courses

Alex Edgcomb[1,2], Frank Vahid[2,1], Roman Lysecky[2], and Susan Lysecky[1]
[1]zyBooks, Los Gatos, California
[2]Computer Science and Engineering, University of California, Riverside
[3]Electrical and Computer Engineering, University of Arizona

ABSTRACT

Small auto-graded coding exercises with immediate feedback are widely recognized as helping students in introductory programming courses. We analyzed usage of a particular coding exercise tool, at 11 courses at different universities. Instructors awarded differing amounts of course points (including zero) for the exercises. We investigated how awarding points affected student completion rates of the exercises. We found that without awarding points, completion rates were about 25%. Awarding even a small amount of points, such as 2 course points out of 100, resulted in 62% completion, with little increase in completion rates for more course points (such as 5, 10, or even 25). Comparing to participation activity completion rates of 85%, one might conclude that the 62% is short of 100% in part due to some students simply not doing homework (15%), and the remaining 23% due to the greater difficulty of the exercises. We analyzed time spent, and found that students spent about 3.3 minutes per exercise, matching the expected 2-4 minutes by the exercise authors. We analyzed number of tries per exercise, and found students submitted 3.5 tries on average per exercise. For some harder exercises, the averages were higher at 5-10 tries, suggesting the students are indeed putting forth good effort. We found very high numbers of tries by some students on a single exercise, sometimes 30, 50, or even 100, suggesting more work is needed to assist such students to better learn the concepts rather than repeatedly submitting tries, and to reduce frustration and increase learning efficiency.

INTRODUCTION

Instructors for introductory programming courses generally agree that having students do small coding exercises on specific constructs, like expressions, assignments, if statements, or loops, helps improve student performance on writing larger programs and on exams [5][12][23]. As such, some schools have developed auto-graded coding exercise tools, and several websites and commercial tools provide such coding exercise tools as well [18][19][20][21][22]. Table 1 includes samples of such coding exercises.

The best way to incorporate such tools into an introductory programming course is still an open question. To help gain insight, we analyzed usage of our interactive learning content [22] across

several universities. Such content includes coding exercises. Questions we sought to answer included:

- How many course points need to be awarded to get students to do the coding exercises?
- How much time are students spending on these exercises, and how many attempts are students making?

BACKGROUND

The state of automated assessment has been documented by Ala-Mutka [15] and Douce [16]. Early automated assessment of programming assignments provided immediate feedback for students [13][14]. Such feedback (including hints [7]) has been shown to be beneficial to the learning process. Kaczmarczyk [10] identified commonly held misconceptions by introductory programming (CS1) students, including memory model representation and default value assignments. Such common misconceptions may be identified during automatic testing of programming homeworks, labs, and assignments.

Researchers [1][2][5][9][17] have analyzed student submission of programming assignments that were automatically graded, wherein the student could submit multiple times per assignment. Spacco [2] collected student programming data across five semesters and three institutions. Spacco found that as students do incrementally-harder exercises, students' scores tended to decrease, whereas the chance of submitting correctly compiling code tended to increase. Dyke [5] recorded 124 students' activity during a programming course with the Eclipse IDE, instrumented with HackyStat [6] for activity collection. Dyke found significant correlations between good programming habits during the assignment and student grades. For example, the amount of use of code generation functionalities (such as, auto-complete) in the code editor was moderately correlated ($r\text{-value} = 0.623$; $p\text{-value} < 0.01$) with each student's average exam grade. Helminen [1] developed a web-based Python development environment that included a coding area and interactive console. 151 students were required to use the environment to complete three programming assignments. 79% of students agreed that the web-based programming environment should be used in the course assignments in the future.

Korhonen found that automated learning systems can provide a comparable learning experience to classrooms in the domain of programming instruction [3]. Korhonen conducted a study with 550 students who used the TRAKLA system [4]. TRAKLA is a web-based environment that individually tailors exercises to the student and provides immediate feedback to students. Korhonen found that the environment was as effective for students as working in a classroom with human tutors giving the feedback.

Students learned more when programming assignments involved solving realistic problems, rather than theoretical problems [8]. Anderson [11] implemented a CS1 course using data analysis for driving examples, such as DNA analysis and predicting election outcomes. Denny [12] conducted an experiment with over 180 students, in which a subset of students invented programming exercises prior to an exam. All the non-inventors solved the exercise. The inventors performed significantly better on the exam and perceived the invention of the exercise as contributing to their learning.

Generally, researchers found that doing more exercises with feedback improves student performance.

OUR CODING EXERCISES

Our coding exercises are integrated into learning content. Our learning content consists of modules, where each module is designed to take 10-15 minutes to complete, not including the coding exercises. Sample modules include "Basic output", "Assignment statements", "Integer division", "For loops", "For loops with arrays", etc. A typical course might cover 10-15 modules per week.

Each module has 2-3 coding exercises at the end, each intended to take about 2-4 minutes to complete. Some basic exercises are designed for 1 minute or less; none are designed to take more than about 4-5 minutes. Sample coding exercise examples are shown in Table 1. The solutions are only visible to instructors.

Table 1: Samples of coding exercises. The # refers to the homework problem from Figure 5. Students do not see the solutions.

#	Instructions	Template	Solution

1	Write a statement that assigns 3 to hoursLeft.	#include <iostream>
		using namespace std;
		
		int main() {
		  int hoursLeft = 0;
		
		  /* Your solution goes here */
		
		  cout << hoursLeft << " hours left." << endl;
		
		  return 0;
		}
			hoursLeft = 3;

3	Write a statement that increases numPeople by 5. If numPeople is initially 10, then numPeople becomes 15.	#include <iostream>
		using namespace std;
		
		int main() {
		  int numPeople = 0;
			numPeople = numPeople + 5;

		numPeople = 10;
		/* Your solution goes here */
		
		cout << "There are " << numPeople << " people." << endl;
		
		return 0;
		}	

5	A cashier distributes change using the maximum number of five dollar bills, followed by one dollar bills. For example, 19 yields 3 fives and 4 ones. Write a single statement that assigns the number of 1 dollar bills to variable numOnes, given amountToChange. Hint: Use the % operator.	#include <iostream>
		using namespace std;
		
		int main() {
		int amountToChange = 0;
		int numFives = 0;
		int numOnes = 0;
		
		amountToChange = 19;
		numFives = amountToChange / 5;
		/* Your solution goes here */
		
		cout << "numFives: " << numFives << endl;
		cout << "numOnes: " << numOnes << endl;
		
		return 0;
		}
			numOnes = amountToChange % 5;

10	Write an expression that will print "Even" if the value of userNum is an even number.	#include <iostream>
		using namespace std;
		
		int main() {
		int userNum = 0;
		
		userNum = 6;
		
		if (/* Your solution goes here */) {
		cout << "Even" << endl;
		}
		else {
		cout << "Odd" << endl;
		}
		
		return 0;
		}
			(userNum % 2) == 0

14	Write a while loop that prints userNum divided by 2 (integer division) until reaching 1. Follow each number by a space. Example output for userNum = 20:
	`20 10 5 2 1`
		#include <iostream>
		using namespace std;
		
		int main() {
		int userNum = 0;
		
		userNum = 20;
		
		/* Your solution goes here */
		
		cout << endl;
		
		return 0;
		}
			while (userNum >= 1) {
			cout << userNum << " ";
			userNum = userNum / 2;
			}

We believe that having students complete a code example, rather than trying to describe all the setup (such as, "Assume int variables x and y have been declared and initialized to 7 and 9, respectively") has a few benefits: Reduces setup explanation necessary, helps students more quickly understand the problem, and allows students to focus on the actual concept being taught. A student can only type on the lines where their solution should appear, thus completing the program.

Furthermore, we have found that providing one concrete input/output example also reduces the amount of setup explanation necessary, and reduces student confusion as to what the problem is asking for.

When a student presses submit, the code is tested using various test cases (usually obtained by our system changing initial values of variables and re-running the program). The student sees each test case, the expected output and the student's output, and whether the code passed or failed that test case. Each exercise is worth two points: One for passing the first test case, and one for passing additional test cases. Typically 2-3 additional test cases exist, designed to avoid "hard coding" of the example's output, and to test border cases).

We intentionally do not limit the number of attempts per exercise. Supporting learning involves a delicate balance between formative assessments and summative assessments, with the balance shifting for different tasks. We view these coding exercises mainly as formative assessments, intended to help the student learn, and not for instructors to ascertain whether the student has learned. As such, we wish to create a "safe" learning environment for students, where they feel free to try, fail, and try again on the coding exercises, without worry of losing points for such trying. (In contrast, for weekly programming assignments, we allow instructors to set submission limits or to meter submission rates).

Note: We do allow instructors to see anonymized student submissions, so instructors can see when students are submitting, how many attempts are being made, and most importantly, what wrong code is being submitted. The latter enables instructors to discuss common mistakes, and common misconceptions, with students during lectures.

HOW MANY COURSE POINTS NEED TO BE AWARDED?

The first question we sought to answer was how many course points should be awarded to encourage students to do the coding exercises. Course points are a limited resource, and instructors thus may want to minimize the course points awarded for formative activities, but at the same time want to use such points to encourage students to do such activities. Most courses have 100 possible course points.

To help answer this question, we searched the web for course syllabi that specified use of our content and that indicated the number of points being awarded for completing the participation and homework activities within our content. We found 11 introductory programming courses having such information publicly available. For each, we could see the number of students using our content. Those 11 courses had the following points (out of 100 course points) and numbers of students:

- 2 courses awarded 0 points (182 students total)
- 1 courses awarded 2 points (55 students)
- 1 course awarded 2.3 points (452 students)
- 2 courses awarded 5 points (382 students)
- 1 course awarded 10 points (31 students)
- 1 course awarded 15 points (56 students)
- 1 course awarded 17 points (99 students)
- 1 course awarded 20 points (115 students)
- 1 course awarded 25 points (33 students)

Total students was 1,596.

To account for different amount of assigned work, the analysis only considered the fourth assigned chapter in each course. We chose the fourth chapter for the following reasons. Chapters in the first weeks of a course are typically easier, and involve students who are enthusiastic early in the term, so results may not be representative. Chapters in the last weeks of a course may involve students who are fatigued, or busy doing high-stakes programming assignments. Because our content is configurable, the fourth chapter may differ for different courses, but is typically branching or loops.

The results of our analysis is shown in Figure 1. If no points are awarded (2 courses), completion rate was about 25%. For some instructors, that number is gratifying, as the students voluntarily completed those activities. But for other instructors, a higher rate is desired.

Assigning course points raised completion rates, as expected. However, somewhat to our surprise, the data shows that the coding exercise completion rates are fairly consistent, averaging about 60%, if *any* number of points are awarded, ranging from 2 to 25 course points out of 100.

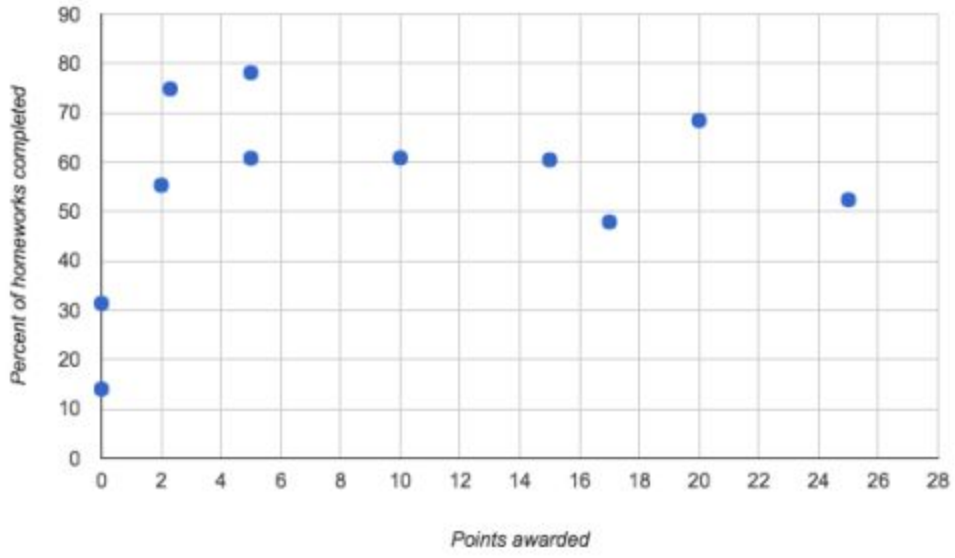
Our own university's course is near 80% completion. We assigned 5 course points out of 100. Our higher rate than other schools may be due to our framing the coding exercise homeworks in a positive manner (explaining the benefits of getting immediate feedback). The higher rate may also be due to how we structure due dates: The module's non-coding activities are due before lecture, and the module's coding exercises are due at the end of the week. We notice that many students attempt the coding exercises before lecture anyway, and then are perhaps more "tuned

in" during lecture, going back and finishing uncompleted exercises afterwards. Our analysis did show some students working for a while on a coding exercise, then a few days later working again (often with numerous attempts in both sessions). We hope to do further analysis of such patterns.

We note that the actual rates are likely a bit higher than shown, due to students dropping a course before or during the fourth chapter, which artificially pulls the rates down. The completion rates for awarding any points was significantly higher (p-value < 0.001) than awarding 0 points.

The main conclusion from the data is that only a few course points need to be awarded to get the achievable completion rate from students.

Figure 1: Percent of chapter 4 homeworks completed across 11 courses. Awarding any number of points caused a jump from about 25% completion to about 60%. Awarding more points had little additional effect.

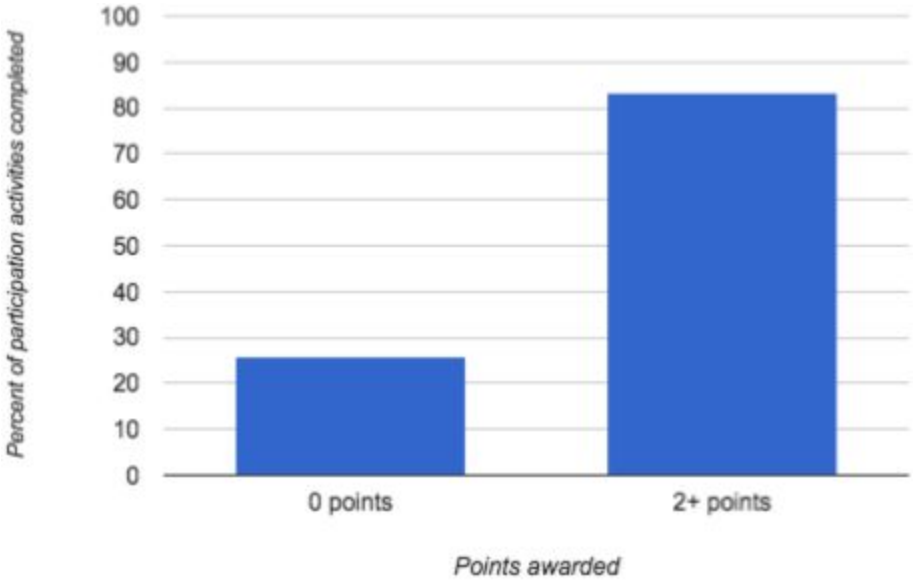


We decided to compare the completion rates for coding exercises with completion rates for the other activities in a module, known as "participation activities". The participation activities are easier, and every student earns all participation activity points just by clicking on buttons. Those activities mostly include answering simple questions in the form of true/false, multiple choice, and short answer (which may involve typing small amounts of code), and for which the correct answers are viewable by the student by clicking a button. The rate for those other activities provide a sort of "upper bound" for the possible rates of the harder coding-exercises.

Figure 2 shows the percent of participation activities completed across 10 courses. One of the courses that awarded 0 points for homeworks awarded some points for participation activities but not a clearly documented amount, so the course was excluded from Figure 2.

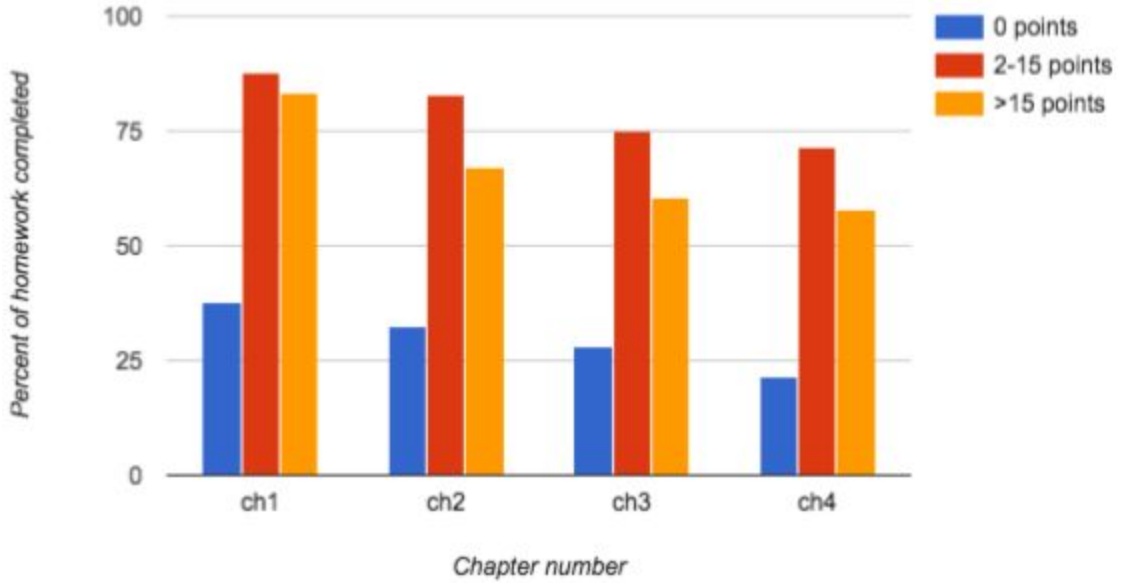
The participation activity completion rate is about 83%, as long as some number of points are awarded. This serves as a sort of "upper bound" for the coding exercises, accounting for students who dropped a course, as well as students who simply will not do homework-like activities. Thus, the difference in completion rates between the participation activities and the coding exercises is likely due to the increased difficulty of the coding exercises, coupled with solutions not being immediately accessible to students. Unfortunately, solutions are improperly made available via some websites, a practice of questionable ethics, but finding those solutions requires some search effort by students.

Figure 2: Percent of participation activities completed across 10 courses The completion rate was about 83%, as long as some course points were awarded.



After conducting the above analysis, we decided to see how the completion rate varied for the first four chapters. Figure 3 shows the percent of homeworks completed, for courses that awarded 0 course points, and courses that awarded any number of course points. The result was that we saw an expected fall in completion rates. When points were awarded, the fall is likely due in part to students dropping the course, subject matter difficulty, workload, and fatigue.

Figure 3: Percent of homework completed across first four chapters. A slight fall is seen, as expected, due to students dropping the course, subject matter difficulty, workload, and fatigue.



HOW MUCH TIME ARE STUDENTS SPENDING, AND HOW MANY ATTEMPTS?

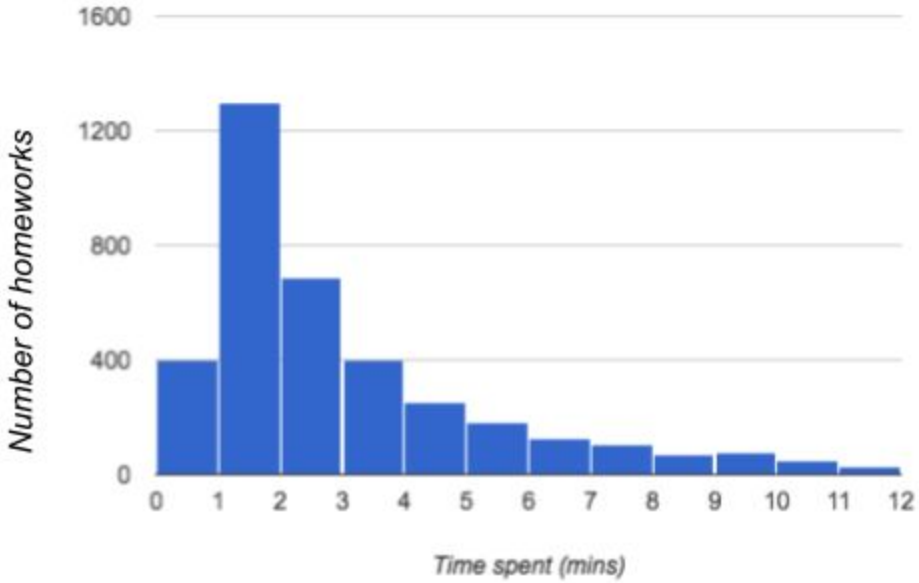
We sought to determine how much time students were spending on these coding exercises. Similarly, we sought to determine how many attempts students were making.

We analyzed one course in introductory C++ programming that had 258 students. The analysis included 16 homework problems from the mid-chapters covered in the course. On average, students completed 92% of those coding exercises.

Time spent was computed by summing the time-between coding exercise submissions. For example, if a student submitted to an exercise at 3:00pm and then re-submitted at 3:02pm, the time-between was two minutes. Further, time spent also sums the time-between a student's preceding participation activity and the first submission to a coding exercise submission. For example, if a student submitted to a participation activity at 4:00pm and then submitted to a coding exercise at 4:06pm, the time-between was 6 minutes. A time-between that was greater than 10 minutes was excluded.

The average time a student spent on homework was 3.3 minutes. As Figure 4 shows, 90% of students spent at least 1 minute on average working on each coding exercise. Many exercises required 1 - 5 lines of code to solve, and each was designed typically to take only 2-4 minutes.

Figure 4: Time spent across 258 students who each completed 16 homework problems (258 students * 16 homework problems = 4128 homeworks). Most students completed within a few minutes. Some students required much more time to complete.



We next analyzed the number of attempts per coding exercise until the student got the correct answer for the first time. So, if a student entered the correct answer on the second try, then there were two attempts. This analysis looked at students who eventually completed the coding exercise. On average, students took 3.53 attempts to complete an exercise, taking 2.77 attempts to earn the first point (first test case passed) and an additional 0.85 attempts to earn the second point (all additional test cases passed). The median number of attempts to complete a coding exercise was 2.

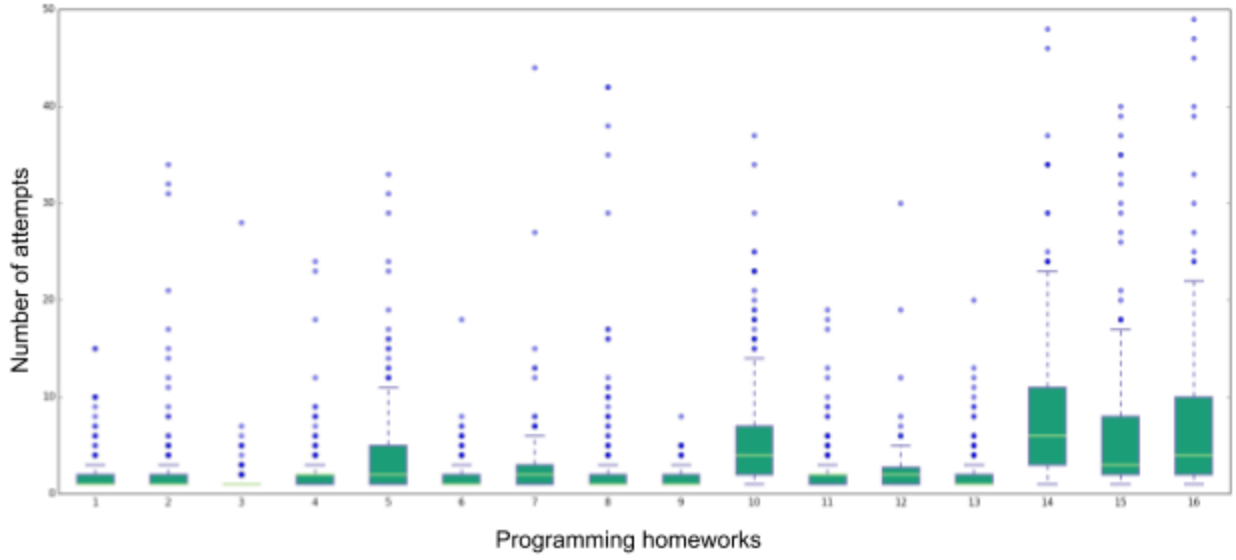
Figure 5 plots each homework problem as a box-and-whisker plot. The low whisker shows the first quartile, the two green boxes show the second and third quartiles, and the upper whisker the fourth quartile. The dots are outliers. For example, for exercise 1, 75% of students completed in fewer than 5 tries, and nearly 100% in fewer than 10 tries, but some students used 15, 20, even 35 tries. One student (not shown) used 70 tries.

The plot shows, generally, that students are trying several times on most exercises and then arriving at a correct solution. However, the plot also shows that some students are either really struggling, or guessing wildly, resulting in 20, 30, or even 40 or 50 tries on one exercise, such as exercises 14, 15, and 16. (In fact, the plot does not show all outliers; some had over 100 tries, with the max being 270 tries by one student on one exercise). Although those outlier numbers show that students are determined, such high numbers are clearly undesirable. We manually examined the code submissions for several cases of high numbers of tries, and found the students

were not wildly guessing, but were certainly making small changes to see if that helped. Often, after 10-20 tries, the code would be rewritten entirely (e.g., a for loop replaced by a while loop) and then changes again made. We hope to further examine those outlier situations and find ways to reduce students struggling, such as giving hints, or possibly metering submissions after some number of tries to encourage the student to think more carefully between submissions.

Despite the outliers, we believe the decision to not limit submissions was appropriate, as students are using the exercises as we had hoped, determinedly trying until getting a correct answer.

Figure 5: Box-and-whisker plot of the number of attempts across 258 students who each completed 16 homework problems. Blue dots are outliers; not all outliers shown; highest is 270.



CONCLUSION

Small auto-graded coding exercises are widely accepted to improve student performance in introductory programming courses. Our analysis shows that only a few course points need to be awarded to get students to put in the effort to do the exercises, as few as 2 course points out of 100, with completion rates of 62%. Awarding more points, such as 10, 20, or even 25, seems to have little additional benefit. As such, we recommend 5 to 10 course points be awarded, since those numbers are less odd-looking than 2 points and leave plenty of room for higher-stakes assessments. Our analysis also shows that students are spending an appropriate amount of time and submitting an appropriate number of tries, although some students are submitting excessive tries of 20, 30, or even 50 or more, suggesting more work is needed to guide such students more efficiently towards learning the concepts.

ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation's Small Business Innovative Research (SBIR) program (1315094 and 1430537), and a Google faculty research award. We are grateful for their support.

REFERENCES

[1] Helminen, Juha, Petri Ihantola, and Ville Karavirta. "Recording and analyzing in-browser programming sessions." Proceedings of the 13th Koli Calling International Conference on Computing Education Research. ACM, 2013.
[2] Spacco, Jaime, Paul Denny, Brad Richards, David Babcock, David Hovemeyer, James Moscola, and Robert Duvall. "Analyzing student work patterns using programming exercise data." Proceedings of the 46th ACM Technical Symposium on Computer Science Education. ACM, 2015.
[3] Korhonen, Ari, Lauri Malmi, Pertti Myllyselkä, and Patrik Scheinin. "Does it make a difference if students exercise on the web or in the classroom?." ACM SIGCSE Bulletin. Vol. 34. No. 3. ACM, 2002.
[4] Korhonen, A., and Malmi, L. "Algorithm Simulation with Automatic Assessment." In Proceedings of the 5th Annual ACM SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education (ITiCSE 2000), ACM, pp. 160–163, 2000.
[5] Dyke, Gregory. "Which aspects of novice programmers' usage of an IDE predict learning outcomes." Proceedings of the 42nd ACM technical symposium on Computer science education. ACM, 2011.
[6] HackyStat. http://csdl.ics.hawaii.edu/research/hackystat/. Accessed: February 2017.
[7] Mohan, Priyanka. Student perceptions of various hint features while solving coding exercises. Dissertation Virginia Tech, 2015.
[8] Parsons, Dale, Krissi Wood, and Patricia Haden. "What Are We Doing When We Assess Programming?." Proceedings of the 17th Australasian Computing Education Conference (ACE 2015). Vol. 27. 2015.
[9] Safei, Suhailan, Abdul Samad Shibghatullah, and Burhanuddin Mohd Aboobaider. "A Perspective Of Automated Programming Error Feedback Approaches In Problem Solving Exercises." Journal of Theoretical and Applied Information Technology 70.1 (2014): 121-129.
[10] Kaczmarczyk, Lisa C., Elizabeth R. Petrick, J. Philip East, and Geoffrey L. Herman. "Identifying student misconceptions of programming." Proceedings of the 41st ACM technical symposium on Computer science education. ACM, 2010.
[11] Anderson, Ruth E., Michael D. Ernst, Robert Ordóñez, Paul Pham, and Ben Tribelhorn. "A Data Programming CS1 Course." Proceedings of the 46th ACM Technical Symposium on Computer Science Education. ACM, 2015.

[12] Denny, Paul, Diana Cukierman, and Jonathan Bhaskar. "Measuring the effect of inventing practice exercises on learning in an introductory programming course." Proceedings of the 15th Koli Calling Conference on Computing Education Research. ACM, 2015.
[13] Saikkonen, Riku, Lauri Malmi, and Ari Korhonen. "Fully automatic assessment of programming exercises." ACM Sigcse Bulletin. Vol. 33. No. 3. ACM, 2001.
[14] Arnow, David, and Oleg Barshay. "WebToTeach: an interactive focused programming exercise system." Frontiers in Education Conference, 1999. FIE'99. 29th Annual. Vol. 1. IEEE, 1999.
[15] Ala-Mutka, Kirsti M. "A survey of automated assessment approaches for programming assignments." Computer science education 15.2: pgs 83-102, 2005.
[16] Douce, Christopher, David Livingstone, and James Orwell. "Automatic test-based assessment of programming: A review." Journal on Educational Resources in Computing (JERIC) 5.3, 2005.
[17] Queirós, Ricardo Alexandre Peixoto, and José Paulo Leal. "PETCHA: a programming exercises teaching assistant." Proceedings of the 17th ACM annual conference on Innovation and technology in computer science education. ACM, 2012.
[18] CloudCoder. https://cloudcoder.org/. Accessed: February 2017.
[19] CodingBat. http://codingbat.com/java. Accessed: February 2017.
[20] Problets. http://problets.org/. Accessed: February 2017.
[21] TuringsCraft. http://www.turingscraft.com/. Accessed: February 2017.
[22] zyBooks. http://www.zybooks.com/. Accessed: February 2017.
[23] Kumar, Amruth N. "Results from the evaluation of the effectiveness of an online tutor on expression evaluation." ACM SIGCSE Bulletin 37.1 (2005): 216-220.