An Investigation of the Conceptual Complexity of Exams Questions in an Introductory Programming Course

Pratibha Menon
menon@pennwest.edu
Department of Computer Science and Information Systems
Pennsylvania Western University, California
California, PA, 15419, USA

Abstract

Instruction in an introductory programming course is typically designed to introduce new concepts and to review and integrate the more recent concepts with what was previously learned in the course. Therefore, most exam questions in an introductory programming course require students to write lines of code that contain syntactic elements corresponding to the programming concepts covered during the instruction. This study investigates the number of concepts involved in the exam problems of an introductory Java programming course. In addition, this study compares how the increase in the number of concepts correlates with the ability of students to write error-free lines of code. The instructional method adopted in this study focuses on providing students with a problem-solving schema and a resultant programming plan that integrates many concepts to meet the problem's goal. Results from this study indicate that as the course progresses through the semester, students, on average, apply appropriate problem-solving schemas and programming plans to produce more error-free lines of code, despite an increase in the concept count in the problems. Furthermore, many exam problems seem to repeat the application of cluster concepts in all three exams. Repeating and building upon the applications of these cluster concepts several times through the course increases the likelihood that students will produce more correct lines of code as the semester progresses.

Keywords: Concepts, Introductory-Programming, Lines-of-code, Exams, Complexity, Program.

1. INTRODUCTION

Failure rates in introductory programming courses have been observed to exceed thirty percent and as a result, there have been efforts to identify causes that make these courses difficult for students (Watson & Li, 2014; Medeiros et al., 2018; Bennedsen & Caspersen, 2019). One thread of research has explored the type of assessment used in introductory programming courses to find out the factors that make the test items difficult (Zur & Vilner, 2014; Ford & Venema, 2010). The difficulty of exam items may be evaluated from student self-assessments or by collecting data on the question attempts. The study presented in this paper infers the difficulty of exam questions by the number of concepts that need to be applied to solve the given problem.

Prior studies have found that the difficulty of exam questions in introductory programming courses varied significantly (Sheard et al., 2013a; Sheard et al., 2013b; Harland et al, 2013) and these questions contained multiple concepts (Goldman et al., 2010). Most of the exams reviewed in these papers are predominantly composed of integrative code-writing questions. Evaluating questions' content and cognitive requirements suggest that students must internalize a large amount of introductory programming content to succeed. While academics can evaluate the difficulty of the exam, they tend to underestimate the total number of

ISSN: 2473-4901
v8 n5777

concepts used versus the ones evaluated. For example, a question on loops requires students to master basic Boolean logic, variables, data types, operators, and syntax before applying these to construct a loop. Some programming concepts are so fundamental that they are used in every code writing instance and must be committed to long-term memory.

Cognitive load theory explains that concepts not fully internalized must be reasoned in the working memory (Ericsson & Kintsch, 1995; Berssanette, & de Francisco, 2022). In addition, the mind is limited in its ability to work with multiple concepts simultaneously, and therefore, students who lack mastery of fundamental concepts face an increasing cognitive burden (Muller, 2007). Results from prior studies argue that we may be asking students in introductory programming courses to master too many concepts in a short time (Goldman et al., 2010).

This study investigates the intrinsic cognitive load of course contents in an introductory programming course by counting the number of concepts used in the exam problems. Furthermore, this study compares the concept count of exam problems with the correctness of the lines of code produced by students for these problems at three different points through a fifteen-week semester. Concept count is considered a metric to evaluate the complexity of an exam problem because instruction in an introductory programming course takes place by introducing students to new concepts or by integrating them with the concepts that were previously taught. The exams are therefore used to assess the conceptual knowledge gained by students and how well they apply these concepts to solve problems.

One of the assumptions of this study is that the exams are preceded by a comprehensive learning process that includes class lectures and a detailed code walkthrough that demonstrates the instructor's practices in applying the concepts to solve problems. In addition, the learning process also includes weekly code-writing assignments that provide practice in applying one or more concepts to various problems. The introductory programming course investigated in this study has three monthly exams that test the ability of students to recall, analyze and apply their conceptual knowledge to write code sequences. Student responses to the exam items are evaluated for their correctness of the expected lines of code. Additionally, the exam questions and student responses are compared to see how the concept count of a problem correlates with the correctness of the lines of code that the students present in the solutions.

An investigation of the problems from three different exams conducted after three intervals in the course are used to reveal how the concept count per problem increases through the course. In addition, an analysis of the correctness of lines of code submitted as exam solutions is used to reveal how well students learn the course contents that involve many interconnected concepts. The results of this study could be used to formulate instructional methods that could be used to teach how these interconnected concepts could be used to solve common computational problems.

2. THE COURSE CONCEPTS AND LINES OF CODE

Course exams are a valuable proxy for deriving curricular expectations and determining what instructors understand as essential. This study explores the concepts used in the exam questions of an introductory Java programming course. While no standard concept inventory exists for introductory programming courses, researchers have used varying methods to derive a list of essential concepts. For example, Tew and Guzdial have used the contents of textbooks to identify ten critical topics (Tew & Guzdial, 2010). Schulte and Bennedsen shortlisted 28 topics from the literature and asked instructors to rate the difficulty and relevance of each (Schulte & Bennedsen, 2006). A prior study by Petersen et. al. evaluated exam contents and observed that the number of concepts considered by instructors while evaluating and grading programming solutions in an exam is fewer than those used to construct the program (Petersen et al., 2011).

This study draws the concepts from a concept inventory created for CS1 courses (Goldman et al., 2010), as depicted in Table 1. All the concepts displayed in this table are treated as being equally difficult, although studies have identified certain threshold concepts that are more critical than other in the learning process (Meyer & Land, 2005; Sanders & McCartney, 2016).

Thus far in computing education, evaluating code complexity has either involved expert evaluation or the use of convenient metrics, such as the number of syntactic elements in a piece of code. While metrics-driven software engineering has fallen out of favor, they are a convenient quantitative method for measuring code (DeFranco & Voas, 2022). One popular metric, lines of code, is commonly used to measure

©2022 ISCAP (Information Systems and Computing Academic Professionals)

ISSN: 2473-4901
v8 n5777

developers' productivity. Lines of code is also an intuitive metric for measuring software size since its effect can be visualized. For example, lines of code could be used to count the volume of instructions (or statements) in a program. Not all lines of code in a Java program may terminate in a semi-colon. For example, a for-loop does not contain a semi-colon but forms a line of code containing an executable entity.

Concept	
Program Structure	Arithmetic Operators
Method Structure	Assignment Operator
Method Parameter	Operator Precedence
Method Return	Proper use of parenthesis
Method Call	Expression
Syntax	Statement
Data Types	Conditionals
Variables	Decision Structures
Literals	Loops
Boolean Operators	Nested Loops
Variable Scope	String methods

Table 1. List of Concepts

Lines of code may be composed differently by novice and professional programmers (Kramer et al., 2017). Skilled developers may be able to apply more syntactic entities with far less code. However, most novice developers, such as the students who attend an introductory programming course, do not pack a physical line of code with too many logical constructs. In fact, while learning, it is easier for the novices to comprehend and write code if each physical line contains the application of fewer logical constructs, and the program is written in a step-by-step manner, as a logical sequence, using separate lines. Therefore, in an introductory programming course, a physical line of code could be used to approximate the program's size.

This study considers two types of heuristics - the number of concepts and the lines of code to study how these metrics could be used to quantify the problem complexity. The lines of code count all the instances of using syntactic elements that correspond to the concepts used to solve the problem. On the other hand, the concept count only considers the distinct concepts used to formulate the solution. This study explores how these "heuristics" could gauge students' learning progress to solve increasingly complex problems in a course during a semester.

3. THE STUDY

This study takes place in a 15-week introductory Java Programming class in an undergraduate-level Computer Information Systems program in a public university. The course has three-unit exams that are spread out through the duration of the course. Exam1 covers the topics of Boolean logic and conditional and decision structures. Students learn about variables and data types and basic input and output statements prior to covering the topic of conditional structures. Exam 2 primarily covers loops, and Exam 3 includes methods. The exams are cumulative as the later exam covers all the concepts assessed in the previous exam. For example, the exam on loops covers Boolean logic, data types, and variables assessed in the previous exam. Exam 3, which includes the topic of methods, also requires students to write algorithms that include loops and conditionals that were previously assessed in Exam 2 and Exam 1, respectively. Therefore, the topics covered and assessed later in the course require students to master the previously covered concepts.

The Exams

The exams were comprised of written response questions of various difficulty levels. Some of these questions are used to evaluate the ability of students to analyze code. Most questions, however, require students to apply their knowledge of programming constructs to write a code solution. The exam questions were of variable points, and scores were assigned to each question based on the correctness of the code lines expected in the solution. In addition, students are given partial points to a solution based on the percentage of the number of lines of code answered correctly. Each hour-long, closed book exam was conducted in a classroom and the exam was strictly timed and proctored. Students access and submit their exams through the course learning management system. Furthermore, due to the time limits of the exams, students were not asked to use a compiler to run their solutions during the exam. The exam's primary intent was to test students' ability to recall and apply their conceptual knowledge to write java statements.

Points carried by an exam question correlated with the number of lines of code students had to write or analyze. Short answer questions required

©2022 ISCAP (Information Systems and Computing Academic Professionals)

https://iscap.info; https://proc.iscap.info

students to write or analyze one or two lines of code. Medium-sized questions had solutions that contained between 5-11 lines of code. More comprehensive solutions that had about 12-26 lines of code. A summary of the characteristics of the exam questions is given in Table 2.

	Exam1	**Exam2**	**Exam3**
Duration	1 hour	1 hour	1 hour
Max points	50	70	100
# of questions	8	8	5
Points/ question	between 5 and 20	between 5 and 20	between 10 and 50
Approx # of expected Lines of Code / question	between 1 and 16	between 1 and 16	between 7 and 25

Table 2 – Exam summary

The Instruction
Before each of the exams took place, students were exposed to the exam topics via class lectures, code demonstrations, practice quizzes, small programming projects, and weekly assignment submissions. Through these learning activities, students are exposed to various problems that apply the concepts listed in Table 1. The code demonstrations covered several application scenarios and code development techniques. As a result, the code walkthroughs served as a method to transfer the instructor's problem-solving schema to the students, who would apply them to solve assignment problems. Every code-walkthrough explained a programming problem and solutions thoroughly using a program plan that reflected the problem-solving schema. The program plan identified the goals and sub-goals of the problems, the concepts, and the lines of code required to meet each sub-goal. In addition, the instruction also explicitly explained how the lines of code that met various sub-goals could be combined to create a program plan. The practice problems and assignments gave an opportunity for students to apply these 'canned' program plans to solve various analogous problems. The exams help the instructor evaluate how correctly students transfer the problem-solving schemas and program plans involving multiple concepts to fit the specific context of the exam problems.

The assignment problems provided means for students to solidify their conceptual understanding and to apply (or modify) their problem-solving schemas to solve similar problems, but from an unfamiliar context. The assignment problems are similar in scope and scale to the ones whose solutions are explained in the code demonstrations.

Exam solutions of 25 students who attended all the three exams were collected. Any information identifying a student was removed from the solutions. Student submissions were not matched across the exams. Students' answers were scored for correctness by comparing them with the expected statements and syntax of the lines of code in the instructor's solutions. Every line of code that formed a statement was checked for correctness and assigned a point only if there were no errors.

4. RESULTS

Composition of Exam Questions
An analysis of the exam questions by the course instructor revealed all the concepts that a student needs to apply to solve each exam question. Figure 1 shows that the total number of concepts increased in the later exams. Figure 1 names each exam problems using the exam and the problem number. For example, E3P5 stands for Exam 3, problem 5. Appendix A shows the mapping of individual concepts to each exam problem. Appendix B lists a partial list of questions from the three exams.

Figure 1 also shows the approximate number of lines of code students were expected to write or analyze in each exam problem. It can be observed from Figure 1 that even a single line of code could contain syntactic elements that represented multiple concepts. For example, the problem E1P1 (described in Appendix B), required students to analyze a statement that contained a compound Boolean expression with comparison and logical operators. While students were evaluated based on their understanding of Boolean expression, they also needed to understand several foundational concepts, such as operator precedence, proper use of parenthesis, and the Java syntax used to comprehend a Boolean expression.

In Exam 1, problems 1, 2, and 3 (depicted as E1P1, E1P2 and E1P3), required students to analyze a given statement, and problems 4, 5, 6, 7, and 8 (depicted as E1P4 – E1P8) required students to write lines of code using the concepts required to write if-else or switch statements.

 ISSN: 2473-4901
v8 n5777

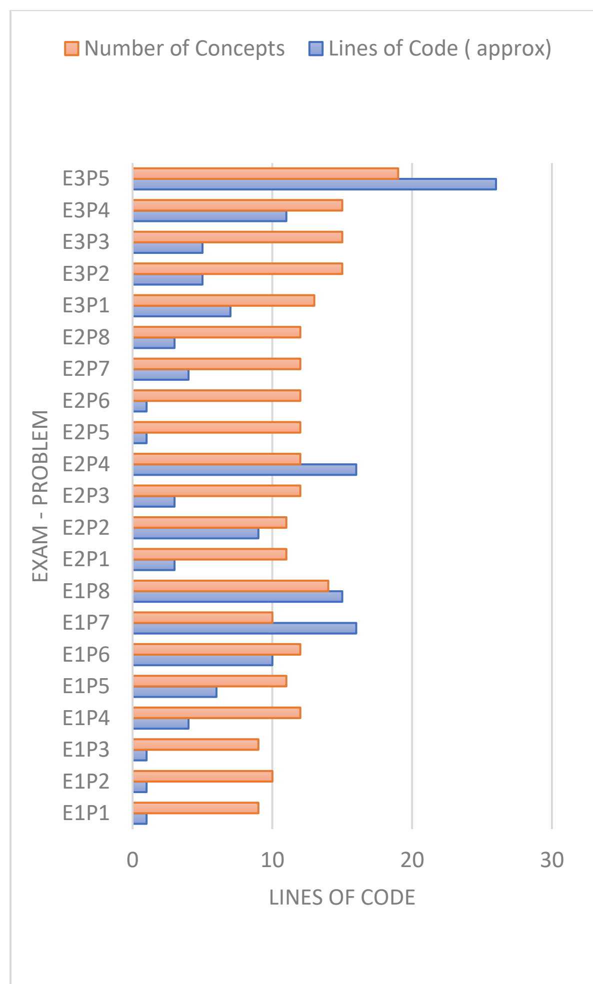

Figure 1. Concept count and the expected lines of code for exam questions.

Appendix B describes some of the questions from Exam1. Writing lines of code that contain decision structures and the actions that follow the truth value of each conditional expression in the decision structure, brings together 10 – 14 concepts, as observed in the concept mapping table in Appendix A. The bars corresponding to E1P5, E1P6, E1P7, and E1P8 in Figure 1, also shows the large number of concepts used to solve these problems.

Exam 2 required students to know how to write applications that use while, do-while, and loops. Programs that included loops also contained foundational concepts such as, variables, Boolean expressions, different types of operators, and simple conditional statements. To apply loops in a program, students also need knowledge of data types and syntax rules to compose expressions and statements. The number of lines of code for the problems E2P1 till E2P8 are shown in Figure 1. Even though two problems may have the same concept count, their lines of code may differ based on the program plan for the solution. Some problems may have additional statements requiring use of a different set of operators and print statements, thereby adding the number of lines of code, without increasing the concept count. For example, this was the case in problem E2P4, when compared to other problems with similar concept count.

Appendix A shows the concept mapping of these problems and Appendix B describes a partial list of the problem statements. The concept count of the Exam 2 problems is relatively high when compared to the expected lines of code in the solutions. For example, writing a simple for-loop requires knowledge of arithmetic and Boolean operators, appropriate use of variables and their scope, and the syntactical elements that are used to compose expressions and statements. Additional concepts are used to write statements that form the loop's body.

Exam 3 required students to write modular code using methods. Students must comprehend the questions and translate the requirements into code by writing the correct return type, arguments, and the statements that go into the method's body. Depending on the problem's requirements, a method's body may require the application of concepts such as arithmetic and Boolean operations, decision structures, or loops. Therefore, questions in Exam 3 also included the concepts that constituted the problems in exams 1 and 2. Appendix A reveal some of the concepts involved in Exam 1 and 2 that were repeated in Exam 3. Figure 1 shows that Exam 3 questions E3P1, E3P2, E3P3, and E3P4 have code lines with high concept counts. The question E3P5, which carried the most points and concept count, required students to write a menu-driven application that repeated several if-else statements to direct the program based on user choices during execution time. Repeating the if-else statements added code lines that used the same concepts, and therefore, the concept count did not increase as much as the code lines did. The table in Appendix B provides a description of problem E3P5.

The Pearson correlation results indicated a significant positive relationship between the lines of code per exam problem and the number of concepts ($r = .592$, $p = .005$). Therefore, for this study, the number of correct lines of code written by students could be used to gauge their latent conceptual knowledge.

©2022 ISCAP (Information Systems and Computing Academic Professionals)
https://iscap.info; https://proc.iscap.info

ISSN: 2473-4901
v8 n5777

Student Performance
Student submissions were scored based on the percentage of the expected lines of code that were correct for each question. Tables 4, 5, and 6 show the average values of correctly written lines of code (or statements) for each question from the three exams.

Problem	Avg Score	# Concepts	Avg Correct lines of code
E1P1	71.41%	9	0.714
E1P2	85.71 %	10	0.857
E1P3	52.38 %	9	0.524
E1P4	68.57 %	12	2.744
E1P5	96.03 %	11	5.76
E1P6	75.24 %	12	7.52
E1P7	58.57 %	10	9.376
E1P8	90.71 %	14	13.605
Avg values	**75.00%**	**10.875**	**5.1375**

Table 4 Exam 1 Results

Problem	Avg Score	# Concepts	Avg Correct Lines of Code
E2P1	82.86 %	11	2.487
E2P2	76.19 %	11	6.858
E2P3	73.33 %	12	2.199
E2P4	75.24 %	12	12.032
E2P5	77.14 %	12	0.771
E2P6	91.43 %	12	0.914
E2P7	86.67 %	12	3.468
E2P8	88.1 %	12	2.643
Avg Values	**79%**	**11.75**	**3.9215**

Table 5 Exam 2 Results

Problem	Avg Score	# Concepts	Avg Correct Lines of Code
E3P1	86.39 %	13	6.048
E3P2	73.47 %	15	3.675
E3P3	85.71 %	15	4.285
E3P4	81.63 %	15	8.976
E3P5	87.66 %	19	22.802
Avg values	**82%**	**15.4**	**9.1572**

Table 6 Exam 3 Results

The tables also show the average percentage score per problem and the number of concepts in each question. The Pearson correlation results indicated a significant positive relationship between the percentage of correct lines of code for each solution and the number of concepts used to solve the problem ($r = .67$, $p < .001$). Results in the tables indicate that problems that required students to apply more concepts were the ones that students tended to score the most. In addition, the tables show that the average test score percentage increased after every exam. If the number of concepts used in a problem indicates its complexity, then this result means students are getting better at handling many concepts as they progress through the course.

Looking at the mapping of concepts in Appendix A, one can observe that the Exam 1 uses many foundational concepts that are re-applied in all subsequent exams. Students also revisit many of these concepts in the assignment problems that follow Exam 1. Many line codes in the second and third exams repeat the concepts used in the first exam. Appendix A also shows that many concepts cluster together in various exam problems. For example, almost all the programs tend to use variables, data types, operators, and expressions. Knowledge of the syntax to construct statements is fundamental to all problems. Introducing newer concepts such as decision structures and loops helps to reinforce the use of foundational concepts covered earlier in the course, such as Boolean expressions and the use of different types of operators. The use of basic operators and inputs and outputs methods reoccur in almost all the programs that involve decision structures and loops. Therefore, repeated application of these concept-clusters to meet the sub-goals of a problem and to create a larger program plan allows students to write correct code involving these concepts in subsequent exams.

A solid understanding of basic concepts and how they occur as a cluster to meet the program’s goal and sub-goals allows students to incrementally integrate newer concepts successfully as they learn to write more extensive and complex programs.

5. DISCUSSION

The positive correlation between the number of concepts used in a code and the correctness of the lines of code indicates students' learning progress in the course. Concept count is directly related to how we currently teach introductory programming courses. Instructional materials such as lectures, code-walkthroughs, and

©2022 ISCAP (Information Systems and Computing Academic Professionals)

ISSN: 2473-4901
v8 n5777

assignments are designed to showcase or practice applications of new concepts or to review and integrate the newer concepts with what was previously learned. As a result, the number of syntactic elements in a code representing the new concepts grows weekly. Furthermore, for novice programmers, adding new concepts to a program requires more code space, resulting in more lines of code.

It must be emphasized that the number of concept-clusters used in a program drives the code size in terms of lines of code and not that the code size will determine the number of concepts used. For example, one could increase the code size by repeating statements with a few concepts. However, having more concepts will require more code space to fit the syntactic elements corresponding to those concepts. The exam questions were created such that questions requiring morel lines of code, also required a program plan that met more sub-goals. Program solutions that required meeting more sub-goals resembles a multi-step problem solving process, where each step involves a cluster of concepts. Therefore, care was taken to ensure that larger code sizes in the exam solutions did not just result from repetition of the similar lines of code that only met one sub-goal.

The Table in Appendix A shows that even writing a simple statement to solve a Boolean or Arithmetic operation requires knowledge of many concepts. For example, nine concepts in the first three questions of Exam 1(E1P1, E1P2, and E1P3) are written using a single line of code that applies operator precedence rules. It is indeed concerning that students must grasp as many as 11 concepts by Exam 1 conducted during week 6 of the course. Even though there is no drastic increase in the number of concepts elsewhere in the course, students must integrate more concepts into their programs as the course progresses.

Suppose the large number of concepts and an increasing need to integrate them into a programming solution adds to students' cognitive load. How were they, based on the results from Tables 4, 5, and 6, able to write more correct lines of code as the semester progressed? One possible explanation is that students may not have reasoned about their lines of code from scratch by thinking about every concept they knew. Most likely, they would have studied the instructor's sample code solutions to develop a problem-solving schema and a programming plan that requires assembling their code using a cluster of concepts.

Classic works in learning theory have argued that learners accumulate schema, or a problem-solving plan, rather than build their solution from scratch by applying all the elementary concepts (Rist, 1989 ; Clancy & Linn, 1999). Per this model, errors in applying a schema to solve a problem in an unfamiliar context or modifying the schema to fit the problem requirements may reveal flaws in the learner's understanding of the concepts used to compose the solution. The application of schema theory to computing pedagogy has taken on a renewed interest, as indicated by the data-mining efforts to study common error patterns encountered by students during their learning process (Zehetmeir et al, 2016).

Repeated schema application or its modification to the problems’ contexts provided opportunities for students to re-apply a cluster of concepts and assemble their solutions multiple times through the course. For example, based on the instructor’s report, decision structure problems E1P5, E1P6, and E1P8 were analogous to problems in previously graded assignments. However, problems E1P1, E1P2, and E1P3 were single-line problems that did not directly resemble any of the assignment problems or were applied as part of a larger program plan. Even though students would have used smaller Boolean expression to build decision structures, problems E1P1, E1P2, and E1P3 needed students to reason about the solution by considering every concept used in the statement. Problem E1P7 was yet another problem that required a considerable modification of the schema applied in the assignments. Similar results were observed in Exam 2, where students scored the most if they could successfully identify similar problems from the assignments and transfer the schema to solve the exam problem.

Students scored the most in Exam 3 because the code inside the bodies of the methods repeated several code schemas previously covered in the assignments and exams. For example, the problems in the final exam required students to apply loop or decision structures in the body of the methods. Students could successfully write the body of the methods in Exam 3 if they learned how to integrate the method concepts with the problem-solving schemas used to solve loops or decision structure problems earlier in the course. A solid application of schemas within the body of a method allowed them to score partial points for a problem, even if they made mistakes in writing the method header or the return statement.

©2022 ISCAP (Information Systems and Computing Academic Professionals)
https://iscap.info; https://proc.iscap.info

ISSN: 2473-4901
v8 n5777

6. CONCLUSIONS

Exams in undergraduate-level introductory programming courses typically assess students' conceptual and syntactic knowledge by requiring them to write lines of code. An evaluation of exam questions reveals that students need to integrate many concepts right from the beginning of the course. As the course progresses, newer concepts are integrated with the ones that were previously covered in the course. Integrating more concepts into the programs leads to the inclusion of more syntactic elements and hence, more lines of code. This study investigates exam solutions submitted by novice programmers and finds a strong positive correlation between the number of concepts used to solve an exam problem and the number of lines of code in the solution. Therefore, the ability of students to apply all the required concepts to write code that solves a problem can be measured by the number of correct lines of code expected in the solution. The correctness of the lines of code and the concept count of problems are two heuristics that could be used to gauge students' progress in the introductory course. Results show that the correctness of lines of code and the concept count increases throughout the code, indicating that students integrate many concepts and concept-clusters in their solutions throughout the course.

The results show that an increase in the concept count may not necessarily reduce the correctness of students' solutions, if they are able to learn the problem-solving schema and create program plans involving multiple clusters. Students and instructors cope with an extensive concept count by clustering the concepts into code patterns corresponding to a problem schema. Rather than reason through the solution concept by concept, students may fit the solution based on familiar patterns of integrating concepts to solve a given type of problem. This finding has implications on how instructional activities could be designed to help students recognize various types of problems 'canned' problem-solving schema that could be applied to solve a variety of analogous application scenarios. Students may then remember each concept, not just isolation, but due to its association with other concepts in a solution's code pattern. This study primarily focused on the ability of students to recall program plans and apply the problem-solving schema to solve exam problems that were like the ones used during the instructional process. Future studies could further investigate the complexity of exam problems based on concept clusters that appear in various problem-solving schema and the ability of students to transfer common problem-solving schema to unfamiliar problems. The difficulty of exam problems could be assessed based on not just the concept count but also conditioned on prior exposure to similar problems.

8. REFERENCES

Bennedsen, J., & Caspersen, M. E. (2019). Failure rates in introductory programming. ACM Inroads, 10(2), 30–36.

Berssanette, J. H., & de Francisco, A. C. (2022). Cognitive Load Theory in the Context of Teaching and Learning Computer Programming: A Systematic Literature Review, *IEEE Transactions on Education*, vol. 65, no. 3, pp. 440-449.

DeFranco, J. F., & Voas, J. (2022). Revisiting Software Methodology. Computer, 55(6), 12–14.

Clancy, M. J. & Linn, M. C. (1999). Patterns and pedagogy. *SIGCSE Bull. 31, 1, 37–42*.

Ericsson, K. A., & Kintsch, W. (1995). Long-term working memory. Psychological Review, 102(2), 211–245.

Ford, M., & Venema, S. (2010). Assessing the Success of an Introductory Programming Course. *J. Inf. Technol. Educ., 9*, 133-145.

Goldman, K., Gross, P., Heeren, C., Herman, G. L., Kaczmarczyk, L., Loui, M. C., & Zilles, C. (2010). Setting the Scope of Concept Inventories for Introductory Computing Subjects. *ACM Transactions on Computing Education*, *10*(2), 1–29.

Harland, J., D'Souza, D., & Hamilton, M. (2013). A Comparative Analysis of Results on Programming Exams. Proceedings of the Fifteenth Australasian Computing Education Conference (ACE2013), Adelaide, Australia.

Kramer, M., Barkmin, M., Tobinski, D., & Brinda, T. (2017). Understanding the Differences Between Novice and Expert Programmers in Memorizing Source Code. Springer International Publishing.

Meyer, J.H.F. & Land, R. Threshold concepts and troublesome knowledge (2005): Epistemological considerations and a conceptual framework for teaching and

©2022 ISCAP (Information Systems and Computing Academic Professionals)

ISSN: 2473-4901
v8 n5777

learning. *High Educ 49, 373–388*.

Medeiros, R. P., Ramalho, G. L., & Falcão, T. P. (2018). A systematic literature review on teaching and learning introductory programming in higher education. *IEEE Transactions on Education*, *62*(2), 77-90.

Muller, O., Ginat, D., & Haberman, B. (2007). Pattern-oriented instruction and its influence on problem decomposition and solution construction. *SIGCSE Bull. 39, 3.*

Petersen, A., Craig, M., & Zingaro, D. (2011). Reviewing CS1 exam question content. *SIGCSE '11*.

Rist, R. S. (1989). Schema Creation in Programming. *Cognitive Science*, *13*(3), 389–414.

Sanders, K. & McCartney, R.(2016). Threshold concepts in computing: past, present, and future. In Proceedings of the 16th Koli Calling *International Conference on Computing Education Research (Koli Calling '16)*. Association for Computing Machinery, New York, NY, USA, 91–100.

Schulte, C., & Bennedsen, J. (2006). *What do teachers teach in introductory programming?* ACM Press.

Sheard, J., Simon, Carbone, A., D'Souza, D., & Hamilton, M. (2013a). Assessment of programming. ACM Press.

Sheard, J., et al., (2013b). How difficult are exams? A framework for assessing the complexity of introductory programming exams.

Tew, A. E., & Guzdial, M. (2010). Developing a validated assessment of fundamental CS1 concepts. ACM Press.

Watson, C., & Li, F. W. (2014). Failure rates in introductory programming revisited. In Proceedings of the 2014 conference on Innovation & technology in computer science education (pp. 39-44).

Zehetmeier, D., Brüggemann-Klein, A., Böttcher, A., & Thurner, V. (2016). A concept for interventions that address typical error classes in programming education. *2016 IEEE Global Engineering Education Conference (EDUCON)*, 545-554.

Zur, E., & Vilner, T. (2014). Assessing the assessment—insights into CS1 exams. In *2014 IEEE Frontiers in Education Conference (FIE) Proceedings* (pp. 1-7). IEEE.

ISSN: 2473-4901
v8 n5777

Appendices

Appendix A

	Exam 1								Exam 2								Exam 3				
Concepts	**P 1**	**P 2**	**P 3**	**P 4**	**P 5**	**P 6**	**P 7**	**P 8**	**P 1**	**P 2**	**P 3**	**P 4**	**P 5**	**P 6**	**P 7**	**P 8**	**P 1**	**P 2**	**P 3**	**P 4**	**P 5**
Method Structure																	x	x	x	x	x
Method Parameter																	x	x	x	x	x
Method Return																	x	x	x	x	x
Method Call																					x
Syntax	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Data Types	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Variables	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Literals	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Boolean Operators	x	x	x	x	x	x		x	x	x	x	x	x	x	x	x	x	x	x	x	x
Arithmetic Operators		x			x	x		x	x	x	x		x	x	x	x	x				x
Assignment Operator	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Operator Precedence	x	x	x	x		x		x													x
Expression	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Statements	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Conditionals				x	x	x	x	x			x			x				x	x	x	x
Decision Structures					x	x	x	x											x	x	x
Loops									x	x	x	x	x	x	x	x					
Nested Loops																x	X	x			
String methods				x								x						x			
Variable Scope													x	x	x			x	x	x	x
Print Methods							x	x	x		x	x	x		x	x			x		x
Input(Scanner) Methods				x				x		x		x						x		x	x

Concepts used in each exam problem for the three exams. The grey cells indicate the main course concept/topic that is evaluated in the problems. Cell that are marked with an 'x' indicate the concept that is used in the lines of code of a correct solution.

ISSN: 2473-4901
v8 n5777

Appendix B

<table>
<tr><th>Exam Problem</th><th>Problem/ Question</th></tr>
<tr><td>E1P1</td><td>What will be the value of boolean var1, which is given as : var1 = (((a*b)<=5)&&(b<1)); if you substitute a = 2 and b = 1 ?</td></tr>
<tr><td>E1P3</td><td>What will be the value of boolean var1, which is given as : var1 = (a.equals("Apple")); if you substitute a = "apple"</td></tr>
<tr><td>E1P5</td><td>Given two input variables : double length and double width. Given one output variable : double perimeter.
Assume that values of length and width are already obtained. Write if statement (just the if statement(s) with the action performed , not the entire program) for the following conditions:
Check if the dimensions are big as follows: If length is greater than 20.0 or width is greater than 20.0, give an output to tell the user that the dimensions are too big.
Check if the dimensions are small as follows: if the length is less than 5.0 or width is less than 5.0, give an output to tell the user that the dimensions are too small.
If the dimensions are neither too big or too small, based on the above two checks - tell the user that the dimensions are within the proper range of values . Then, calculate the area = length * width and print the value of that perimeter.</td></tr>
<tr><td>E1P8</td><td>Write a program that obtains from the user the age of a child in months. The program will determine the required next vaccinations based on the given age. Write a complete program that will look at the given age in months and determine the next vaccination required. You may skip commenting your code for this exam to save time. You program should compile and be logically and syntactically correct.
<table>
<tr><th>Baby's Age in months</th><th>Next Vaccination</th></tr>
<tr><td>0</td><td>HepatitisB</td></tr>
<tr><td>Greater than 0, less than 6 (inclusive)</td><td>DaTP</td></tr>
<tr><td>Greater than 6, less than 12 (inclusive)</td><td>MMR</td></tr>
<tr><td>Greater 12</td><td>"Will complete the program later on"</td></tr>
</table></td></tr>
<tr><td>E2P2</td><td>In this problem you will write a loop in which you ask users to enter the price of an item and add that price to the total price. In your loop, you will ask the user to enter a 1 to “Scan” and a 2 to “quit”. Loop until the user types 2 - to quit.
For as many times, as the user enters a 1- to "Scan an item' : ask the user for the price of the items , and add this price to a variable called totalPrice.
The variables price and totalPrice are both doubles .Assume that the Scanner object is already declared and named as input.
Declare any extra variables that you have used in your loop - other than price, totalPrice or input.</td></tr>
<tr><td>E2P6</td><td>The for loop shown below loops several times and produces a final value of i that is used to calculate the value of j. However, there is an error : int j = i % 5 ; //This statement shows an error : "cannot find symbol i"
Errored code:
for(int i = 1; i<100; i=i*5){
System.out.println (i);
}
Rewrite the code above so that it fixes the error given in the error statement</td></tr>
<tr><td>E3P3</td><td>Define/Write a method called codeThePlayer that takes two argument – an integer called playerID and a String called playerName . This method has a void return. If the playerID value is equal to 100, the method prints out the following : "Admin ID ". Else, if the playerID is not a 1, the method prints out the playerName, followed by the statement: "Not Admin".</td></tr>
<tr><td>E3P5</td><td>Menu driven program
Write a program called pointsCalculator that calculates the total points earned by using a credit card for travel and hotel stays. The program provides the user with the following menu :
“ Enter 1 to select mileage points”
“Enter 2 to select a hotel points”
“Enter 99 to quit”
If the user enters a 1 , ask the user to enter the mileage (which will be a double type). Scan the mileage and call a method called calculateMileage that takes in as mileage as a parameter. This method returns a double value to be stored in a variable called mileagePoints. Print out the value of mileagePoints.
If the user enters a 2, ask the user to enter the number of hotel stays(which will be an integer type). Scan the hotel stays and pass this variable as a parameter to the method calculateHotelPrice . This method returns a double variable called hotelPoints. Print out the values of hotelPoints.
Assume the methods are already defined - you just need to call them in the code. You also don't need to implement a while loop.</td></tr>
</table>

Some of the Exam Problems from Exam 1, 2 and 3