On Learning How to Program via an Interactive eBook with Adaptive Parsons Problems

by

Carl Christopher Haynes-Magyar

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Information)
in the University of Michigan
2022

Doctoral Committee:

Assistant Professor Barbara Ericson, Chair
Professor Barry Fishman
Associate Professor Maya Israel
Professor Kevin Miller
Assistant Professor Steve Oney

BETHLEHEM ACADEMY
1992 - 1993
NURSERY R
MS RODGERS

Carl Christopher Haynes-Magyar

cchaynes@umich.edu

ORCID iD: 0000-0002-9637-6285

DEDICATION

To my mother, Sharon Dupree Haynes, who said I could do anything I set my mind to.

ACKNOWLEDGMENTS

I want to acknowledge the nice and the good advisors, researchers, teachers, and peers that helped me solved all the problems that led me to this moment.

Bethlehem Academy (1993): Kathy Rogers

The Ethan Allen Public School 306 (1995): Mr. Silvent

Frank D. Paulo Intermediate School 75 (2000): Andrea Obermaier, Arthur Kaufman, Arthur Siegel, Carolyn Rega, Charles Oliveri, Charlyn Dickey, Denise Korchak, Elizabeth Kotite, Ellen Chan, Ira Schwartz, Leonard Lichter, Linda Quintavalle, Peter Guinta, Peter Viani, Sandra Hofferman, Steven Martinson, Trinita Hourican, and Linda Kane

Tottenville High School (2006): Clarence J. Jones, Cliff Bloom, Eileen O' Connell, John P. Tuminaro, Mara Bhaerman, Miss Ohlsen, Mr. Lopez, Mr. Tierney, Mrs. Biunno, Mrs. Fernandez, Robert Torres, Sal Annarumma, Stefania Alvaro, Steve Rubinson, Thomas Warren, Vincent Corvino, and Will Cicoria

College of Staten Island The City University of New York (2010): Cate Marvin, Debra Evans-Greene, Lee Papa, Matt Brim, Mazen Naous, Peter Keil, Phil Sigler, Sarah Schulman, Terry Rowden, Timothy Gray, Tyehimba Jess, and Wilma L. Jones

Syrcuse University (2016): Anna Chovanec, Barbara Stripling, Blythe Bennett, Brian Dobreski, Bryan Semaan, Carsten Østerlund, Chad Harper, Daniel Cohen, David Lankes, Deborah Nosky, Jason Dedrick, Jill Hurst-Wahl, Katrina Maust, Laura Cooper, Lauren Juiliani, Margaret Craft, Megan Oakleaf, Michael Kicey, Renee Hill, Robert Heckman, and Sheila Clifford-Bova

University of Michigan (2022): Barbara Ericson, Barry Fishman, Christopher Brooks, Elizabeth Yakel, Erin Krupka, Kentaro Toyama, Laurie Buis, Libby Hemphill, Lionel Robert, Mark Guzdial, Mustafa Naseem, Nicole Ellison, Paul Edwards, Paul Resnick, Robin Brewer, Sarita Schoenebeck, Stephanie Teasley, Soo Young Rieh, Steve Oney, Tawanna Dillahunt, Thomas Finholt, Tiffany Veinot, Adriene Beltz, Allison Ryan, Aadarsh Padiyath, Allison Sweet, Allura Casanova, Charles

Senteio, Charlotte Beaudoin, Christina Costa, Emma Flores, Evan Hoye, Feranmi Okanlami, Hari Subramonyam, Ihudiya Williams, Ioulia Kovelman, Jasmine Jones, Katie Cunningham, Megh Marathe, Mez Perez, Olga Panteleeva, Phillip J. Bowman, Richard Gonzalez, Robert Adams, Veronica Falandino, William Dering, Zihan Wu

International Computing Education Research (ICER) Doctoral Consortium (2020): Colleen Lewis, Lauren Margulieux

Access Computing - University of Washington: Amy J. Ko, Brianna Blaser, Richard E. Ladner

Google Computer Science Research Mentorship Program (2021): Sau Man (Samantha) Lo

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE

LIST OF TABLES

LIST OF APPENDICES

LIST OF ACRONYMS

ADHD Attention Deficit Hyperactivity Disorders

AI Artificial Intelligence

CER Computing Education Research

CS Computer Science

CS CLCS Computer Science Cognitive Load Component Survey

CSEd Computer Science Education

DSM-V Diagnostic and Statistical Manual of Mental Disorders, Fifth Edition

HCI Human-Computer Interaction

IEP Individualized Education Program

LEM Learning Edge Momentum

MSLQ Motivated Strategies for Learning Questionnaire

SDL Self-Directed Learning

SRL Self-Regulated Learning

SSD Office of Services for Students with Disabilities

WAI Web Accessibility Initiative

ZDP Zone of Proximal Development

ABSTRACT

Novice programmers need well-designed instruction and assessment informed by research and critical perspectives to conquer the historical challenges associated with completing introductory computer programming courses successfully. These issues include high dropout and failure rates, the struggle to acquire and retain basic programming knowledge, and bias and stereotype threat due to social markers (race, gender, dis/ability, sexuality, etc.).

Unfortunately, traditional programming practice, such as code writing, can be arduous, time-intensive, and frustrating. Adaptive Parsons problems, which require learners to place mixed-up code blocks in the correct order and indentation, are designed to support learners' individual differences in knowledge acquisition, reduce extraneous cognitive load, and improve affect while learning how to program. These problems modify the difficulty of the current or next problem based on a learner's prior performance and help-seeking behavior. Adaptive Parsons problems are a more interactive way to learn while using worked-examples of stereotypical solutions to programming problems; learners acquire strategies for arranging and creating these solutions. Hence, they can help novice programmers build up the kind of mental library of solutions experts have at their disposal when writing code from scratch to solve any number of critical problems related to computing.

This multi-manuscript presents studies aimed at the exploring the problem-solving efficiency of Parsons problems that optimize cognitive load as a substitute for traditional computer programming practice. Mixed methods are used to understand how learners think, behave, and feel when learning how to program via an interactive eBook with adaptive Parsons problems and equivalent write-code problems. First, I conducted field experiments to evaluate the design of these problems for active learning during lecture. Second, I redesigned these problems and tested hypotheses about cognition and learning to understand cognitive, behavioral, and affective learning outcomes impacted by these design changes. And third, I explored access and equity issues for neurodiverse learners.

First, results showed undergraduates are significantly more efficient at solving a Parsons problem versus an equivalent write-code problem, but not when the solution to the Parsons problem was uncommon (not the most common student written solution). And, while most students (80.6%) reported finding Parsons problems useful for learning, some students with prior programming experience expressed strong negative reactions to them. This led to the development of a feature to toggle between a Parsons problem and an equivalent write-code problem. Second, I confirmed the following hypotheses about Parsons problem solutions. Novice programmers were significantly more efficient at solving a Parsons problem created with the most common student written solution versus writing the equivalent code. And, when first presented with a Parsons problem that had an uncommon solution, learners tended to use that solution to solve an equivalent write-code problem. Third, I note four observations about accessibility for learners with seizure disorders, ADHD, mental health disabilities, and memory impairment. Cross-synthesis analysis evidenced that participants benefited from readings in the eBook that chunked information into smaller chapters and sections.

This research has implications for the creators of computer programming practice problems who are tackling historic issues related to underrepresented populations in computing by engaging in a critical analysis of how to provide adaptive scaffolding for all learners. Novice programmers with and without disabilities, who require extended time to retain information, benefit from increasing the efficiency and quality of knowledge acquisition, positive attitudes about assessments, and the accessibility of interactive programming practice environments.

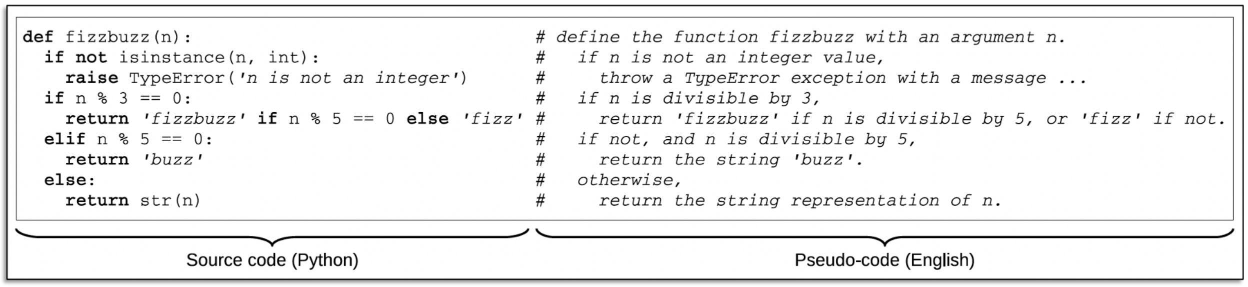
CHAPTER 1

Introduction

Learning how to program can be difficult [90]. It is an interactive process requiring the development of several complex skills [311]. Computing education theorists posit these skills include: code reading and tracing, code writing, pattern comprehension, and pattern application [222, 416]. Learners are often shocked as they encounter systems that ask them to learn all of these skills simultaneously [90, 200]. But through explicit and incremental instruction, novice programmers *can* develop these skills [20, 416].

Elliot Soloway, a pioneer in computer science education (CSEd), suggests new programmers be taught explicitly about "stereotypical solutions to programming problems as well as strategies for coordinating and composing them" [356, p. 850]. However, the acquisition and retention of these skills (i.e., academic growth or improvement and expertise) depend heavily on the quality and quantity of deliberate practice (i.e. where practice is focused on improving specific tasks) [7, 106] and the cognitive, metacognitive, affective, behavioral, and cultural factors that may influence learning how to program [111, 294, 311, 388]. The problem is that current introductory computer programming instruction and assessment fail to scaffold and/or sequence these skills [416] and fail to be critical—make visible the links between computing and injustice [194].

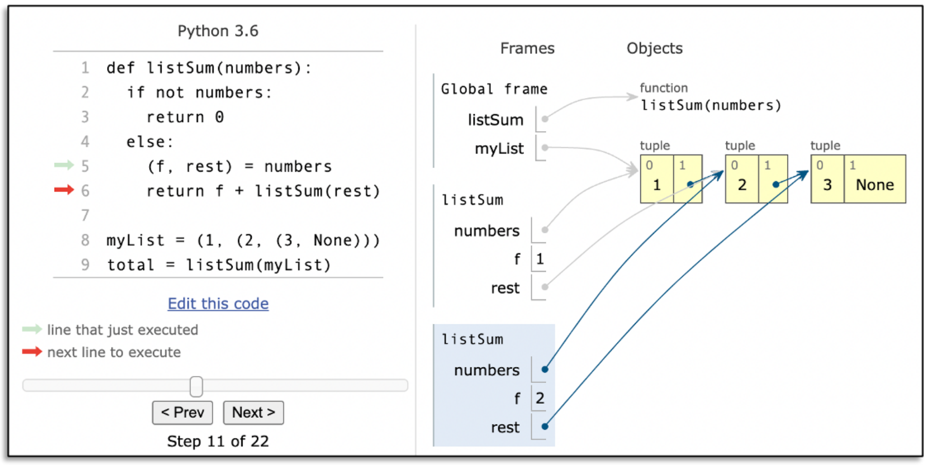
Traditional introductory computer programming practice has included writing pseudocode, code tracing, and code writing [22, 309, 337, 342, 396]. Pseudocode is a plain language description of the steps in a program (see Figure 1.1) [268]. Creators of introductory computer science assessments have used pseudocode to test students' understanding of fundamental computing concepts [379]. Pseudocode can also reduce the cognitive complexity of programming tasks [22]. Students who solve programming problems with pseudocode or subgoals (i.e., predetermined procedures for solving problems) perform significantly better than those who do so without pseudocode or subgoals [254]. Pseudocode can be helpful because novice programmers don't need to worry about the syntactic details of a programming language. Code tracing (see Figure 1.2) involves using paper and pencil or online tools such as PythonTutor [132] to trace the execution of a program [202]. Some learners prefer writing pseudocode or subgoals to code tracing [72, 255]. And although code writing is an authentic task because it resembles real-world practice [338], it requires learners to write code from scratch. This can be can be time-intensive, frustrating, and can decrease students' engagement and motivation which is one of the root causes of struggle in early computer science courses (see Figure 1.3) [23, 187, 327, 344]. Unfortunately, not much has changed in the last decade [327].



```
def fizzbuzz(n):                                      # define the function fizzbuzz with an argument n.
  if not isinstance(n, int):                          #   if n is not an integer value,
    raise TypeError('n is not an integer')            #     throw a TypeError exception with a message ...
  if n % 3 == 0:                                      #   if n is divisible by 3,
    return 'fizzbuzz' if n % 5 == 0 else 'fizz'       #     return 'fizzbuzz' if n is divisible by 5, or 'fizz' if not.
  elif n % 5 == 0:                                    #   if not, and n is divisible by 5,
    return 'buzz'                                     #     return the string 'buzz'.
  else:                                               #   otherwise,
    return str(n)                                     #     return the string representation of n.
```

$\underbrace{\hspace{6cm}}_{\text{Source code (Python)}}$ $\underbrace{\hspace{7cm}}_{\text{Pseudo-code (English)}}$

Figure 1.1: *Example of Pseudocode with Python (left) and Corresponding Pseudocode Written in English (right)*, from [268, p. 575].



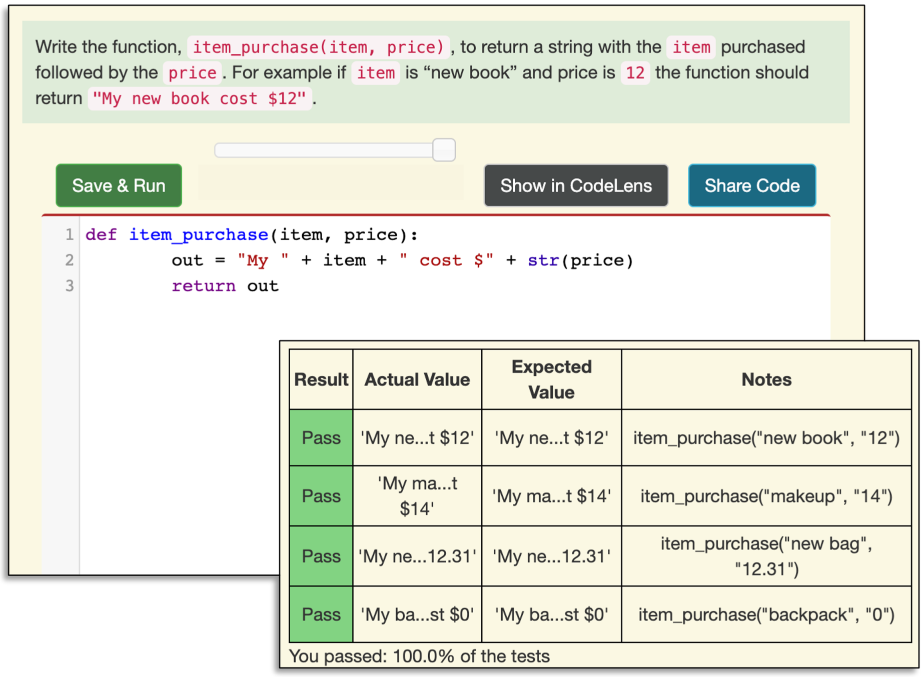
```
def listSum(numbers):
  if not numbers:
    return 0
  else:
    (f, rest) = numbers
    return f + listSum(rest)

myList = (1, (2, (3, None)))
total = listSum(myList)
```

Figure 1.2: *Example of code tracing with Python Tutor. It depicts the eleventh step of executing a function,* ***`listSum(numbers)`****, to unpack the tuple numbers.*

Other causes of struggle in early computer science courses include personal obligations, a lack of sense of belonging, and a lack of confidence [256, 327, 363]. These courses often have poor retention and typically see high dropout rates from students [26, 27, 311]. Learners' acquisition and retention of basic programming knowledge are still fragile after these courses [311]. And, more importantly, the discipline struggles with the recruitment and reflection of society writ large concerning social markers (i.e., race, gender, dis/ability, sexuality, etc.) [256, 311, 363].

When problem-solving, some novice programmers experience high cognitive load because they lack the necessary schemata or plans to guide them [311]. Their unfamiliarity with the semantic and syntactic errors they make may cause them frustration [208, 228, 311]. Learning how to write


Write the function, `item_purchase(item, price)`, to return a string with the `item` purchased followed by the `price`. For example if `item` is “new book” and price is `12` the function should return `"My new book cost $12"`.

```
def item_purchase(item, price):
        out = "My " + item + " cost $" + str(price)
        return out
```

Result	Actual Value	Expected Value	Notes
Pass	'My ne...t $12'	'My ne...t $12'	item_purchase("new book", "12")
Pass	'My ma...t $14'	'My ma...t $14'	item_purchase("makeup", "14")
Pass	'My ne...12.31'	'My ne...12.31'	item_purchase("new bag", "12.31")
Pass	'My ba...st $0'	'My ba...st $0'	item_purchase("backpack", "0")

You passed: 100.0% of the tests

Figure 1.3: *Example of a code writing problem. It depicts a function to return a string with an item purchased and its price and reveals four unit tests that it has passed.*

code and recognize these errors can be overwhelming because this process imposes demands on working memory and affects how information is stored in and retrieved from long-term memory [311]. Programming practice problems with high element interactivity material “consist of elements that heavily interact and so cannot be learned in isolation” [368, p. 124]. These problems require students to have amassed knowledge of various concepts to complete them successfully [227]. This can create *un*desirable difficulties [57]. Computing education researchers argue that highly interrelated (or high element interactivity) programming practice does “not give students the opportunity to show which concepts they understand and which they do not” [227, p. 286].

Furthermore, programmers from historically underrepresented groups in computing typically have less prior programming experience, which can lead to underperformance [6, 236, 233]. They are more vulnerable to being negatively affected by poorly designed learning environments and difficult programming tasks that are introduced without the appropriate support [175, 183, 220]. In general, computer science undergraduates are vulnerable to negative self-assessments [130]. Repeated failures, especially when first learning, lower self-efficacy for computer programming [187]. Women, Black, Latinx, Native American, and Pacific Islander (BLN+) students report in-class confusion on material, assignments, and/or programming bugs significantly interfere with their performance versus non-BLN+ students [327]. Moreover, for learners with disabilities, inadequate support for their cognitive differences may dissuade them from learning how to program

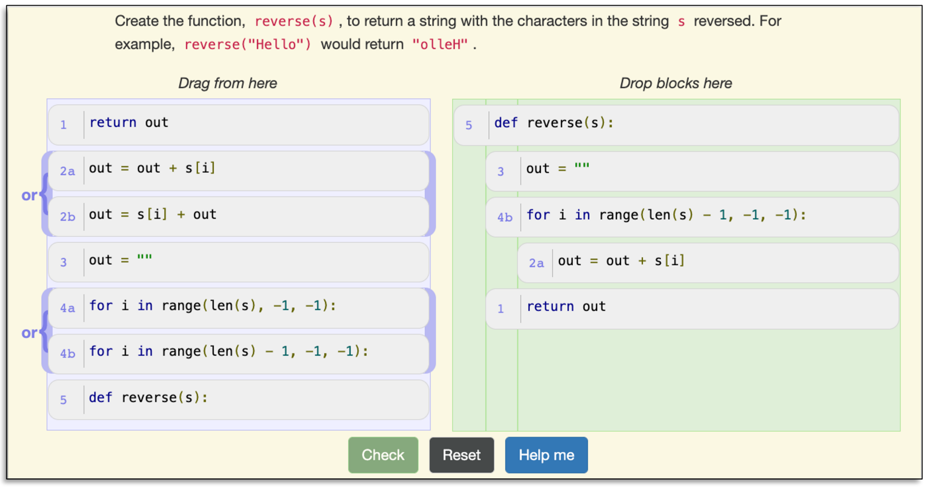
[76, 367]. Computing courses are increasingly being offered online [298] and within some online programming environments, accessibility barriers and challenges exist for learners with disabilities, such as unclear content, limited interactivity, and difficulty with multistep problem-solving [172, 197, 211, 247, 257, 1].

Novice programmers need well-curated instruction and assessment informed by equity, diversity, and access initiatives to conquer these and other historical issues associated with enrolling in and completing introductory computer programming courses successfully. Such instruction and assessment should scaffold learning to provide support to learners as they develop programming skills and reduce that support gradually as their ability increases [133, 304]. Robins proposed the Learning Edge Momentum (LEM) hypothesis to explain how people learn to program and the high element interactivity of programming concepts [308]; he links this hypothesis to research in education on scaffolding, learning transfer, analogy, and the Zone of Proximal Development (ZPD)—the difference between what learners can do autonomously and what they can do with support [397]. The theoretical foundation of the Learning Edge Momentum (LEM) hypothesis is based on the following principle:

> “We learn ‘at the edges’ of what we already know by adding to existing knowledge. The more that new information is given a meaningful interpretation (i.e., the richer and more elaborate the links between new and old knowledge), the more effective learning appears to be... [311, p. 351]

Mixed-up code (Parsons) problems, also called Parsons Programming Puzzles [283] provide the kind of scaffolding that can keep learners in the Zone of Proximal Development (ZPD) [89, 100]. Parsons problems are drag-and-drop practice exercises that require learners to place blocks of code in the correct order and sometimes require the correct indentation as well. They are used to introduce novice computer programmers to introductory computer programming concepts (see Figure 1.4). In this dissertation, I compare how learners solve adaptive Parsons problems to writing the equivalent code from scratch.

Parsons problems, also considered completion problems [102, 387], are an explicit way to help novice programmers learn about stereotypical solutions to programming problems and strategies for arranging and creating them [356, 405]. Hence, Parsons problems may help novice programmers build up the kind of mental library of solutions experts have at their disposal when writing code from scratch to solve any number of critical problems in education, energy consumption, government, and healthcare for example [194]. Block-based practice exercises like this help counter orthodox views of programming as difficult; they are learning tasks designed to challenge but not frustrate students [98]. These kinds of learning tasks can improve diversity by supporting learners with a broad range of academic abilities and prior experience [170, 215]. Historically underrep-


Create the function, `reverse(s)` , to return a string with the characters in the string `s` reversed. For example, `reverse("Hello")` would return `"olleH"` .

Drag from here

```
1   return out
    or
2a  out = out + s[i]
2b  out = s[i] + out
3   out = ""
    or
4a  for i in range(len(s), -1, -1):
4b  for i in range(len(s) - 1, -1, -1):
5   def reverse(s):
```

Drop blocks here

```
5   def reverse(s):
3       out = ""
4b      for i in range(len(s) - 1, -1, -1):
2a          out = out + s[i]
1       return out
```

Figure 1.4: *Example of a mixed-up code (Parsons) problem. It asks the programmer to create a function called* **`reverse(s)`** *to return a string with the characters reversed.*

resented minorities and females perform better on problems that are block-based versus writing code from scratch [407, 406]. Novice programmers often solve them more efficiently than writing the equivalent code. This saves them time when problem-solving which can empower them to try solving more problems [102, 100, 425]. Struggling students and students with disabilities, who often require extended time to retain information, benefit from increasing the efficiency and quality of knowledge acquisition while problem-solving [161]. Parsons problems can be used as formative and/or summative assessments in the classroom [79, 102]. They can be used as an active learning strategy during lecture and outside of the classroom via eBooks. The following studies are the first to investigate their use in these two contexts (see [21] for a review on research related to learning how to program outside the classroom). Active learning techniques cause learners to engage in meaningful cognitive learning tasks that engage rather than remain cognitively passive [see 62]. Parsons problems can improve motivation and learning for students from underrepresented groups [195, 380]. And finally, drag-and-drop programming platforms are increasingly being used by companies to create programs in the absence of professional developers [415] thereby supporting the use of Parsons problems to teach programmers working in low/no-code environments. I explore the following research questions:

- What is the effect on problem-solving efficiency, affective, and behavioral factors of solving adaptive Parsons problems versus writing the equivalent code for neurotypical and neurodiverse learners?
- What are neurotypical and neurodiverse programmers' practices, perspectives, and attitudes regarding an interactive eBook with adaptive Parsons problems?

In summary, mixed-up code (Parsons) problems model good code, manage cognitive load, provide immediate feedback, maximize engagement, allow students to demonstrate partial knowledge, and help teachers identify concepts students are struggling with [79, 89, 283]. In this dissertation, I explore, in more depth than previous research, factors that influence how neurotypical and neurodiverse learners (those with cognitive, learning, and/or neurological disabilities) learn to program using a interactive eBook embedded with dynamically adaptive Parsons problems [99, 103]. Dynamically adaptive Parsons problems are Parsons problems that modify the difficulty of the current or next problem based on a learner's prior performance (inter-problem adaptation) and help-seeking behavior (intra-problem adaptation) [100]. By investigating these questions, I aimed to explore the following thesis statement:

By investigating these questions, I aim to prove my thesis statement:

> **By exploring the effects of solving adaptive Parsons versus write-code problems, it may be possible to create adaptive learning environments that are more effective, efficient, and inclusive of the perceptions of learners with and without dis/abilities learning in and outside of the classroom.**

1.1 Adaptive Parsons Problems for Active Learning

In the current and following subsections, I will briefly describe the main four parts of this dissertation/multi-manuscript.

To motivate the use of adaptive Parsons problems for active learning in introductory computer programming courses, we (1) conducted three between-subjects experiments to test the effect on efficiency (time to correct solution) of solving adaptive Parsons problems with distractors versus writing the equivalent code and (2) disseminated an end-of-course survey to understand undergraduate students' attitudes towards adaptive Parsons problems as lecture assignments. All prior research studies were either in a controlled environment with volunteers or in a lab (not lecture). The research questions were:

RQ1: What is the effect on efficiency (time to correct solution) of solving adaptive Parsons problems with distractors versus writing the equivalent code when used as lecture assignments?

RQ2: What are undergraduate students' attitudes towards adaptive Parsons problems as lecture assignments?

The median time to solve each Parsons problem was less than the median time to write the equivalent code for all but two of the problems, both with complex conditionals. However, that difference was significant for only six of the ten problems. Our hypothesis for why there was a higher median for the time to solve two of the Parsons problem than writing the equivalent code was that the problem instructions did not match the Parsons problem solution and/or they also had a large number of possible correct solutions. Results from student surveys also provided evidence that most students (78%) find solving adaptive Parsons problems in lecture helpful for their learning, but that some (36.2%) would rather write the code themselves. These findings have implications for how to best use Parsons problems.

1.2 Problem-Solving Efficiency and Cognitive Load of Adaptive Parsons Problems

Driven by the insights obtained from the study above, we then conducted a within-subjects experiment to compare the efficiency and cognitive load of solving adaptive Parsons problems versus writing the equivalent code. The prior study showed that solving adaptive Parsons problems was significantly more efficient than solving isomorphic write-code problems, however, it used a between-subjects design. This study sought to test the hypothesis that self-reported cognitive load is less for solving Parsons problems than for isomorphic write-code problems [100, 98]. Hence, we used a within-subjects design "because the measurement of cognitive load is made independently of individual differences that would otherwise corrupt a between-subjects design" [408, p. 3]. We also performed a behavioral analysis using a think-aloud protocol with open-ended interview questions. The goal was to gain a deeper understanding of what programmers were thinking as they solved a problem set with both adaptive Parsons and write-code problems, their preferences and understanding of intra-problem adaptation. The research questions were:

RQ3: What is the effect on efficiency of solving adaptive Parsons problems with distractors versus writing the equivalent code?

RQ4: What is the effect on cognitive load of solving adaptive Parsons problems with distractors versus writing the equivalent code?

RQ5: How does efficiency relate to cognitive load?

RQ6: What are undergraduate students' attitudes towards adaptive Parsons problems versus write-code assessments?

Based on the results from four out of five problems, 69 to 92% of the time, undergraduates were significantly more efficient at solving Parsons problems versus solving equivalent (isomorphic) write-code problems. The difference was significant for four of the five problems, but not for the problem with the highest mean cognitive load rating. The solution to this Parsons problem was uncommon to students in that it used a `while` loop; most learners who solved the equivalent write-code problem first used a `for` loop. And, interestingly enough, there was an ordering effect. Seventy-seven percent of the students who solved that problem as a Parsons problem first, used its solution when solving the equivalent write-code problem. Results varied in significance for the difference between mean cognitive load ratings of the two problem types and the relationship between cognitive load ratings and efficiency for the two problem types. Participants also reported write-code problems to be harder than Parsons problems. However, they felt that they learned more from write-code problems if they did not get stuck while trying to solve them. These findings have implications for how to best generate and sequence (order and select) Parsons problems for learning transfer.

1.3 *(Un)*Common Parsons Problems Solutions

Novice programmers should be explicitly taught stereotypical solutions to programming problems [356]. The prior study on problem-solving efficiency and cognitive load revealed that a Parsons problem with an uncommon solution was not significantly more efficient to solve than writing the equivalent code. Hence, we hypothesized that changing that Parsons problem to the most common student written solution would make it significantly more efficient to solve. To test our hypothesis, we conducted a mixed within and between-subjects experiment with ninety-five undergraduates. We also explored the relationship between problem-solving efficiency, cognitive load ratings, self-efficacy, and clusters of write-code solutions. And finally, previous research also found that students were confused when pressing the "Help Me" button provided indentation and combined blocks that were already adjacent (see the section on Parsons Problems in Chapter two). Therefore, the adaptation process was modified to no longer provide indentation and to combine blocks that were the furthest apart. To understand how students solve Parsons problems, in particular the problem with an uncommon solution, and the impact of changing the adaptation process, we report on three think-aloud observations with undergraduates. The research questions were:

RQ7: What are the effects on efficiency of solving adaptive Parsons problems created from the most common student written solution or an uncommon solution versus writing the equivalent code? What are the order effects?

- **H1:** If a Parsons problem with an unusual solution is modified to use the most common student written solution then students will be more efficient at solving it.
- **H2:** If students are first presented with a Parsons problem that has an uncommon solution then a high percentage will use that solution to solve an equivalent write-code

problem.

RQ8: What is the effect on self-reported cognitive load ratings of solving adaptive Parsons problems versus solving equivalent write-code problems?

RQ9: How does problem-solving efficiency relate to self-reported cognitive load ratings and self-efficacy beliefs?

RQ10: Do students find the modified intra-problem (same problem) adaptation process understandable and useful?

RQ11: Why did students struggle to solve the Parsons problem with an uncommon solution?

The results confirmed our hypothesis and its inverse. Students were significantly more efficient at solving a Parsons problem with a common solution and students used an uncommon Parsons problem solution to solve an equivalent write-code problem significantly more efficiently which resulted in higher learning gains. There were also significant positive correlations between efficiency and cognitive load and significant negative correlations between efficiency and self-efficacy. And finally, results revealed that students who struggled to solve the Parsons problem with an uncommon solution could benefit from help with (1) planning, (2) self-regulated learning, and (3) more explanation of distractors. Also, students did not report any new problems due to modifications of the adaptation process. These findings have implications for how to automatically generate adaptive Parsons problems and improve the adaptation process.

1.4 Accessible Adaptive Parsons Problems

Adaptive programming practice problems can improve problem-solving efficiency, lower cognitive load, and most undergraduate novice programmers find them useful for learning. But these are results from studies focused on neurotypical individuals learning how to program *inside* the classroom [104, 102, 100, 98, 153]. Relatively little research in computing education focuses on learners with cognitive disabilities [76, 198] and learning how to program *outside* of the classroom [21]. Hence, we know little about whether these drag-and-drop/block-based computer programming practice problems are accessible to novice programmers with cognitive, learning, and/or neurological disabilities when learning is self-paced.

To explore the accessibility of adaptive Parsons problems for neurodiverse learners, how they learn when using an interactive eBook, and affective factors such as attitudes about computing and self-efficacy, I conducted an exploratory multiple case study. One of the goals was to observe participants during think-aloud sessions and generate working hypotheses for future investigation. This study was based on a proposal I modified after receiving feedback and making changes due to the pandemic [152]. The research questions were:

RQ12: What are the accessibility barriers/challenges and benefits reported when neurodiverse learners use an interactive eBook to learn how to program?

RQ13: What are their computational practices, perspectives, and attitudes?

RQ14: What are their perceptions of the usability and usefulness of adaptive Parsons problems versus write-code problems for learning how to program?

RQ15: What are their perceptions about the surveys and/or questionnaires used in the study?

The first research question led to observations for working hypothesis and future research [421] for each participant about how to improve the accessibility of adaptive Parsons problems and interactive eBooks for learning how to program. First, for learners who experience seizures there is a positive correlation between solving programming problems with numeric calculations and seizures. Second, there is a negative correlation between germane cognitive load (i.e., the working memory resources devoted to learning) and the number of distractor blocks in a Parsons problem for learners with an attention deficit hyperactivity disorder (ADHD). Third, when presented with media computation programming problems that involve modifying, editing, and/or creating pictures of contextualized and culturally relevant content, then learners with an attention deficit hyperactivity disorder (ADHD) and/or mental health disabilities will experience more positive emotions. And fourth, active learning strategies such as note-taking will improve problem-solving performance for learners with memory impairments.

Findings across all the participants led to several design recommendations based on accessibility barriers/challenges and benefits. One barrier/challenge for all of the participants was the length and structure of some of the reading and practice assignments. Participants also reported a need for peer-support. Results suggested learners need help developing self-directed learning and self-regulated learning skills, taking on intellectual challenges, maintaining a growth mindset, and experiencing a sense of belonging. And nonbinary/genderqueer participants took issue with binary survey/questionnaire items.

All of the participants reported benefiting from readings in the eBook that chunked information into smaller chapters and sections. One of the main benefits to learning outside of the classroom was the interactivity of the eBook and the flexibility to complete the reading and practice assignments at ones own pace. Participants also reported no negative reactions to frequently filling out self-assessments and concerns about comparison information concerning how much mental effort peers invest in solving programming problems. And, finally, most participants found the Parsons problems useful for learning, however two participants had trouble with indentation and one had trouble with the combine blocks feature.

1.5 Contributions

My dissertation contributes to the research on how novice programmers with and without cognitive, learning, and/or neurological disabilities learn to program using adaptive Parsons problems and equivalent write-code problems in an interactive eBook both in and outside of the classroom. These studies are the first to (1) study the use of adaptive Parsons problems during lecture, (2) explore learners' problem-solving efficiency and cognition in depth as they practice programming, (3) extend research into learners' understanding of intra-problem adaptation, (4) explore the use of common and uncommon Parsons problem solutions for efficiency and learning transfer, (5) investigate the accessibility of adaptive Parsons problems, and (6) explore the computational practices, perspectives, and attitudes of neurodiverse learners learning to program outside of the classroom.

1.6 Thesis Outline

Chapter 2 reviews related work. Chapter 3 covers one between-subjects study that uses adaptive Parsons problems as an active learning pedagogical technique during lecture. Chapter 4 explores the efficiency and cognitive load of solving adaptive Parsons problems versus equivalent write-code problems through a within-subjects study. Chapter 5 tests hypotheses about adaptive Parsons problems with common and uncommon solutions for efficiency and learning transfer. Chapter 6 extends prior research on adaptive Parsons problems to neurodiverse populations to generate working hypotheses for future research. And, lastly, Chapter 7 concludes this dissertation, summarizes key takeaways, and discusses future work.

CHAPTER 2

Related Work

In this section, I review the literature related to several areas: 1) Parsons problems, 2) active learning, 3) efficiency and time-on-task, 4) cognitive load, 5) self-efficacy, and 6) neurodiversity.

2.1 Parsons Problems

Parsons Programming Puzzles, also called mixed-up code (Parsons) problems are drag-and-drop practice exercises that require learners to place blocks of code in the correct order and may also require indentation. In 2006, Dale Parsons and Patricia Haden developed them for introductory programming courses to: maximize engagement, help students learn syntax, introduce common errors, model well-written code, and provide instant feedback [283]. These types of problems enable learners to demonstrate semantic and strategic knowledge without having to generate syntax [254], although some students use syntactic clues within the blocks to piece together the solution without necessarily understanding the problem which can lead to a trial-an-error approach to solving—an important pedagogical design challenge for creators of Parsons problems to solve [79, 425]. These problems typically only have one correct solution while there are many ways to write code from scratch [283]. They prompt the kind of explicit learning Soloway advocated for [356]. Explicit learning is facilitated by direct and unambiguous delivery of procedures and scaffolding to guide learners through the learning process with clear goals and ways to measure success [9]. Instructors use Parsons problems as formative and/or summative assessments [79, 102]. Scores on Parsons problems correlate highly with scores on write-code assignments [60, 79]. In their groundbreaking study, Parsons and Haden asked undergraduates to solve Parsons problems at the start of each lab for an introductory programming course and results showed that 82% of the students rated the problems as useful or very useful for learning Pascal [283]. Since that initial study, researchers have developed a variety of Parsons problems. Parsons problems vary by dimension, feedback, adaptation, and use of distractors [89, 98, 404].

2.1.1 Dimensions

Parsons problems can be *one-dimensional* or *two-dimensional*. One-dimensional Parsons problems require learners to put blocks of mixed-up code in the correct order [283, 79]. Two-dimensional Parsons problems require learners to put code blocks in both the correct order and also provide the correct indentation as shown in the solution for Figure 1.4 [165, 182]. Indentation is used to determine the grouping of code in some programming languages. There is evidence that two-dimensional Parsons problems are harder than one-dimensional Parsons problems [79]. Studies show that student opinions about solving two-dimensional Parsons problems ranged from boring to fun [156] and that students with more prior programming experience do not find them as useful as students with less experience [155]. However, a log file analysis showed more learners tried to solve two-dimensional Parsons problems than nearby multiple-choice questions in an interactive eBook, which is additional evidence that most learners find them useful for learning [101].

Furthermore, an experiment under controlled conditions using a between-subjects design provided evidence that undergraduate students solve two-dimensional Parsons problems with distractors significantly more efficiently than fixing code with errors or than writing the equivalent code with similar learning gains from a pretest to a posttest [102]. And further research provided more evidence that two-dimensional Parsons problems are more efficient for learning than writing the equivalent code [100]. Likewise, Zhi et al. found that students solved block-based Parsons problems in Scratch in about half as much time as students solving write-code problems in the lab with no negative effect on later work [425], but these experiments were all done in controlled settings in a lab.

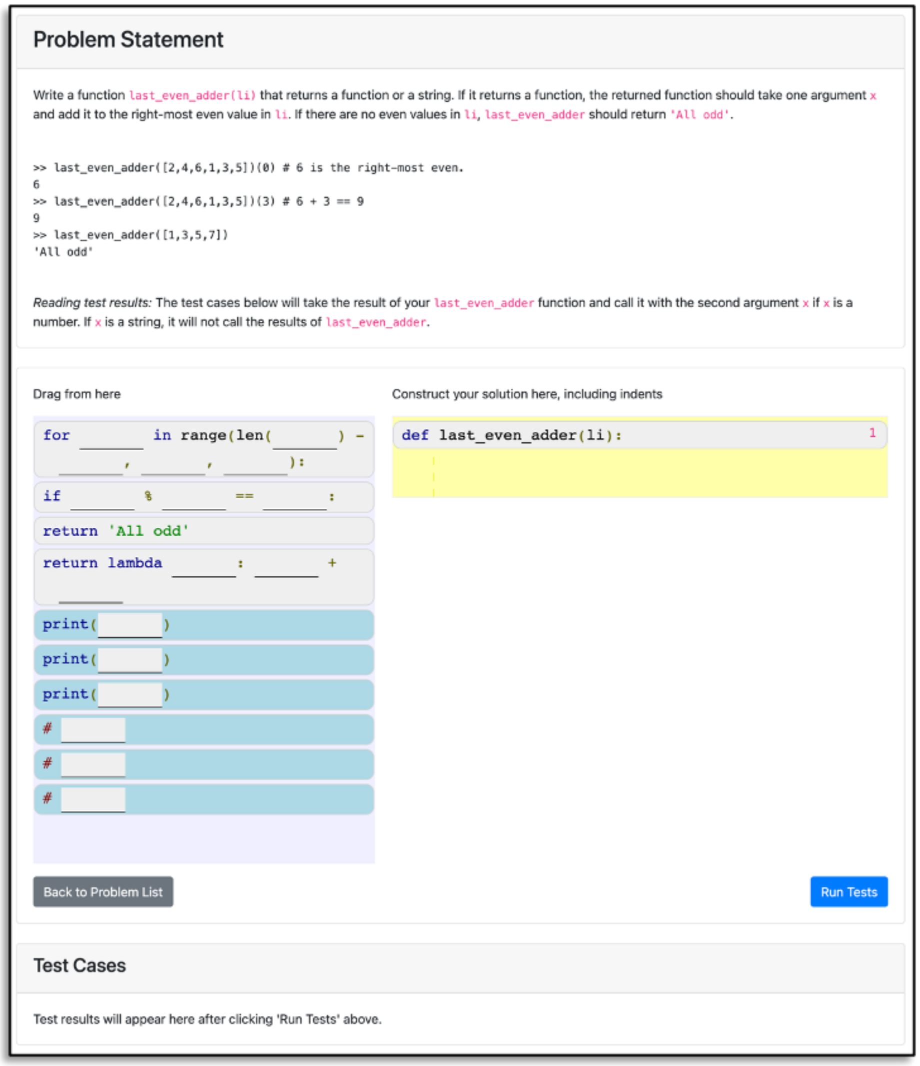
2.1.1.1 Subgoal Labels or Pseudocode

Subgoal labeling is a technique used to help increase learners' problem-solving skills by having them break down the solutions into specific goals [53]. It is akin to pseudocode, which programmers use to convey the specific steps of an algorithm in plain English so that anyone with basic programming knowledge can understand. Programmers refer to pseudocode as "informal textual representations of a program or algorithm" [22, p. 227]. Solving one-dimensional Parsons problems with provided subgoal labels has led to significantly more learning gains for understanding how to write a **`while`** loop than solving Parsons problems with self-generated or no subgoal labels [254].

2.1.1.2 Faded Parsons Problems

A recent variation of Parsons problems, called Faded Parsons problems, asks learners to rearrange and complete blocks of code, with partially blank lines, into a valid program (see Figure 2.1)

[404, 405]. They require learners to put code blocks in the correct order, provide the correct indentation, and fill in blanks with the correct code. The goal is to fade some of the scaffolding that Parsons problems typically provide by asking learners to partially write code from scratch. A pilot study provided evidence that students perceived this type of problem as more difficult than a typical Parsons problem, but less difficult than writing the equivalent code [404]. Faded Parsons problems are significantly more effective for teaching pattern comprehension and pattern application over code writing and code tracing exercises and comparably improve students' ability to write code from scratch [405].


Problem Statement

Write a function `last_even_adder(li)` that returns a function or a string. If it returns a function, the returned function should take one argument `x` and add it to the right-most even value in `li`. If there are no even values in `li`, `last_even_adder` should return `'All odd'`.

```
>> last_even_adder([2,4,6,1,3,5])(0) # 6 is the right-most even.
6
>> last_even_adder([2,4,6,1,3,5])(3) # 6 + 3 == 9
9
>> last_even_adder([1,3,5,7])
'All odd'
```

Reading test results: The test cases below will take the result of your `last_even_adder` function and call it with the second argument `x` if `x` is a number. If `x` is a string, it will not call the results of `last_even_adder`.

Drag from here

```
for ______ in range(len(______) -
  ______, ______, ______):
if ______ % ______ == ______:
return 'All odd'
return lambda ______: ______ +
  ______
print(______)
print(______)
print(______)
# ______
# ______
# ______
```

Construct your solution here, including indents

```
def last_even_adder(li):
```

Test Cases

Test results will appear here after clicking 'Run Tests' above.

Figure 2.1: *Example of a Faded Parsons Problem. It asks the learner to complete a function.*

2.1.2 Feedback

Researchers have also studied different ways to provide feedback on Parsons problems: *execution-based* or *line-based*. Execution-based feedback systems run the code in the constructed solution and indicate correctness, return expected and actual values, and/or return any error messages [155].

Line-based feedback systems highlight a code block to indicate it is in the wrong position or change the background color to indicate that the solution is correct [182]. A study comparing the two types of feedback provided evidence that line-based feedback led to quicker solutions than execution based feedback [155]. Line-based feedback may be easier for novice learners to understand than compile-time error messages. One weakness of line-based feedback systems is their ability to facilitate learners' adoption of a trial-and-error approach [89]. However, researchers have reported this type of feedback is used sparingly [156]. The Parsons problems in these studies provided line-based feedback.

2.1.3 Adaptation

Adaptive practice can improve learning, take less time, and increase engagement compared to non-adaptive learning [68]. Furthermore, adaptive learning strategies, such as help-seeking, can provide assistance from more knowledgeable individuals or computers when students "recognize difficulties they cannot overcome on their own" [265, p. 315]. Computing education researchers have explored help-seeking behavior of novice student programmers and the different reactions to receiving human-based and/or computer-based help [295]. Help-seeking behavior is impacted by three categories of factors (1) inputs (i.e., students' previous experiences, expectations of the tutor, and independence), (2) student mindset (i.e., trust in a tutor's ability and recognition that one needs help), and (3) the attributes of help (i.e., specificity and interpretability) [295]. Price et al. found computer-based help does not threaten help-seeking behavior in the same way that human help does and that computer-based help is "efficient, salient, and accessible" [295, p. 133]. They suggest designers of adaptive learning systems consider the importance of facilitating help-seeking and of evaluating the quality of computer-based help.

Soloway, Guzdial, and Hay called for a change in Human-Computer Interaction (HCI) from User-Centered Design to Learner-Centered Design [358]. In particular, they called for more scaffolding that supports learners as they try to accomplish a new task [316]. They describe several types of scaffolding including limiting the starting task so that it is not overwhelming, modeling behavior, providing hints, encouraging reflection, and encouraging metacognition. To be most effective, scaffolding should fade as the learner develops expertise. In other words, the system should adapt to the learner's performance and provide them with hints and feedback. However, as the authors say, *"Build learner-centered software! Easy to say, hard to do"* [358].

There have been several efforts to make Parsons problems adaptive and evaluate help-seeking features. One Parsons problem system uses selection adaptation which means that it selects the next problem from a pool of Parsons problems based on a learner's prior performance and randomly generates distractors [203].

2.1.3.1 Intra- and Inter-Problem Adaptation

Ericson [100] developed two types of adaptation for Parsons problems to keep students in Vygotsky's Zone of Proximal Development (ZPD) [397]. This zone represents the difference between what learners can do autonomously and what learners can do with support [397]. This means that the learner is challenged to do more than they could accomplish without help. The adaptability and adaptivity [see 414] of the system is designed to support learners' individual differences in knowledge acquisition, support optimal cognitive load, and improve affect (i.e., emotions and self-efficacy while learning how to program) [100, 98]. The goal is to maintain desirable difficulties (a tasks requiring just the right amount of effort) and reduce or eliminate undesirable difficulties during programming practice [102, 100, 419]. It was built on the premise that adaptation increases learning efficiency and engagement [68].

Intra-problem (same problem) adaptation is learner-initiated; it occurs when the learner clicks the "Help Me" button which then removes a distractor, provides indentation, or combines two blocks into one until only three blocks are left. [98]. This help is only available after learners submit at least three incorrect solutions.

Inter-problem (between problem) adaptation is system-initiated; it occurs when the system modifies the difficulty of the next Parsons problem based on the learner's preceding Parsons problem performance [98]. It does this by removing distractors and pairing distractors with the correct code (making it easier) or by adding distractors and jumbling them with correct code blocks (making it harder) [98]. If the learner solved the last Parsons problem in just one attempt the next one is made harder.

Secondary teachers and undergraduate students solving two-dimensional adaptive Parsons problems reported finding them helpful for learning to fix and write Python; learners are nearly twice as likely to correctly solve adaptive Parsons problems than non-adaptive Parsons problems [98].

In contrast to these types of adaptation, intelligent tutoring systems can be described as having an inner loop and an outer loop [394]. The inner loop executes feedback and hints during a task while the outer loop uses information from the inner loop (i.e., performance on the task) to select the next task. This is known as problem-sequencing [201, 394].

In the following studies, I use both intra-problem and inter-problem adaptive Parsons problems; however, only the first Parsons problem would have been affected by the learner's past performance on Parsons problems since the inter-problem adaptation takes place before the page is loaded and all of the problems were displayed on the same page.

2.1.4 Distractors

Distractors are code blocks that are not needed in the correct solution. Distractors expose learners to common misconceptions about syntax or logic [283]. Distractors can be paired or jumbled. Problems with paired distractors display the correct and incorrect code blocks adjacent to each other. Jumbled distractors, on the other hand, are randomly mixed in with the correct code blocks. Parsons problems with as many distractors as the number of correct code blocks have been reported as overwhelming for learners [79]. Studies have shown that distractors make Parsons problems more difficult and increase self-reported cognitive load [79, 146].

However, controlled experiments have provided evidence that undergraduate students can solve Parsons problems with a limited number of distractors significantly more efficiently than writing the equivalent code [102, 100]. Researchers have hypothesized that distractors can be used to help beginners learn to recognize and fix common syntax and semantic errors [283]. One qualitative study provided evidence that secondary teachers felt that solving Parsons problems with distractors helped them learn to fix and write-code [98]. However, further research is necessary to confirm these results, determine the self-reported cognitive load of both solving Parsons problems and write-code problems, and dig deeper into what students are thinking while solving both types of problems [89].

Parsons problems with and without adaptation impact affective, behavioral, and cognitive (ABC) learning outcomes [79, 89, 102, 100, 98, 153, 283, 312, 405, 339, 425]. Cognitive learning outcomes correspond to changes in cognitive abilities and resources (e.g., efficiency or time-on-task and cognitive load), behavioral learning outcomes correspond to changes in engagement, study skills, etc., and affective learning outcomes correspond to changes in attitudes and motivation (e.g., self-efficacy) [see 315]. With regard to cognitive learning outcomes, studies provide evidence it is significantly more efficient to solve Parsons problems with adaptation than to solve equivalent write-code problems [339, 425] and equally as effective for pre-posttest learning gains [100]. However, researchers suggest further investigation into the effectiveness of Parsons problems [89] and their impact on self-reported cognitive load [100, 98]. Previous research has also shown Parsons problems increase behavioral learning outcomes such as engagement [82, 101, 283, 339] and affective learning outcomes such as interest, motivation, and self-efficacy [82, 98, 339].

In summary, there is evidence that Parsons problems can improve problem-solving efficiency, lower cognitive load, maximize engagement, and help teachers identify where students are struggling [79, 89, 283, 312, 425]. Yet, “Parsons problems can be perceived as difficult because they require students to read code written by others (using syntax and logic that might not be in their personal comfort zone)" [79, p. 7]. Furthermore, some advanced learners want harder programming problems while novices struggle with the exact same problems [101, 104, 102]. And given the challenges of developing learning technologies that Human-Computer Interaction researchers have identified [362] and the fact that learners' attitudes about assessments influence educational outcomes [44, 42], it is important to examine student perceptions of Parsons problems.

This dissertation explores the use of adaptive Parsons problem for programming practice and its impact on problem-solving efficiency, cognitive load, self-efficacy, programming strategy use, and both learners' practices, perspectives, and attitudes. It is different in that it explores (1) the use of adaptive Parsons problems in an uncontrolled setting (a classroom) as an active learning pedagogical technique, (2) the use of different instruments to measure cognitive load and self-efficacy, (3) the impact of common and uncommon Parsons problem solutions on efficiency and transfer, and (4) the practices, perspectives, and attitudes of neurodiverse and neurotypical learners.

2.2 Active Learning

Active learning refers to a variety of learning approaches, such as solving problems or peer instruction, where the student is active rather than passive. Several studies have shown that active learning results in more learning and a lower failure rate than passive learning, such as a traditional lecture [118, 138]. Active learning also increases student engagement and motivation [47, 56, 125, 126, 139, 157, 158, 177, 184, 196, 225, 231, 249, 252, 261, 280, 287, 290, 324, 329, 334, 340, 341, 400], improves students' learning experiences and performance [47, 56, 64, 125, 126, 131, 139, 143, 157, 158, 177, 196, 206, 212, 231, 252, 261, 269, 287, 324, 329, 333, 334, 340, 341, 400, 429, 428], encourages student interest, involvement, and engagement [47, 64, 125, 157, 252, 261, 269, 280, 290, 329], promotes the development of soft skills [64, 139, 157, 177, 225, 252, 333], promotes flexibility and self-paced learning [56, 64, 184, 249, 329, 400], increases students' confidence [47, 139, 252, 324, 329], helps teachers optimize classroom time [56, 184, 249, 329], and connects struggling students to high performing students [249, 333].

The ICAP theory defines student engagement in terms of what can be observed [61, 62]. It describes four modes: interactive (I), constructive (C), active (A), and passive (P) and theorizes that I $>$ C $>$ A $>$ P for learning. An example of passive learning is receiving information without doing anything else, such as in a traditional lecture. For a behavior to be deemed active there must be some physical motion and focused attention. Constructive behavior occurs when the learner creates something beyond what is in the learning materials, such as a concept map. Interaction requires two individuals or a learner and a computer agent engaging in a constructive discussion with turn taking.

Note that if students are taking verbatim notes during a traditional lecture, they are being active. If students are writing notes in their own words, they are being constructive. If students are discussing comprehension questions with a partner, they are being interactive.

Active learning pedagogical techniques used in computer programming courses include [30]: flipped classrooms [13, 56, 64, 75, 184, 206, 212, 225, 231, 249, 290, 324, 329, 333, 340, 341, 400], project-based learning [107, 139, 157, 261, 334], peer instruction [47, 131, 429, 428], blended learning [56, 269, 341], collaborative learning [157, 177, 252], problem-based learning [47, 56, 225], game-based learning [126, 158], pair programming [249, 252], hands-on learning [139], inquiry-based learning [225], living code [340], peer-teaching with videos [287], POGIL [417], team-based learning [212], and think-pair-share [196]. Some of the difficulties encountered when using active learning pedagogical techniques to teach computer programming include: the significant effort required of teachers to execute active learning pedagogical techniques [47, 56, 107, 157, 225, 287, 329, 341, 417], the resistance of students to engage in active learning pedagogical techniques [13, 75], and the significant effort required of students during active learning [212].

In chapter three, this dissertation explores the use of adaptive Parsons problems as a type of active learning activity during lecture to add to existing active learning pedagogical techniques. Adaptive Parsons problems require learners to drag code blocks into the correct order. They require both physical motion and focused attention so they meet the definition of active learning in the ICAP framework. While students could also be given write-code problems to work on during lecture, some of them would likely get stuck and need individual help [23, 187], which is hard to provide in a typical lecture auditorium when teachers' time is already limited. Since adaptive Parsons problems can reduce the difficulty of a problem by removing distractors, providing indentation, or combining blocks, even struggling students can reach a correct solution which can increase engagement and decrease resistance to active learning. Prior research has shown that students are nearly twice as likely to correctly solve an adaptive Parsons problem than a non-adaptive one [98].

2.3 Efficiency and Time-on-Task

Time is a limited human resource and it can take some students twice as long to learn what other students learn [35]. Spending more time learning the same material to achieve the same performance as one's peers can also leave learners feeling frustrated and decrease their motivation to learn [35].

Educational researchers both within and outside of computing education have used the term ‘time-on-task’ to describe the amount of time students spend on learning [35, 214, 320, 360]. Both efficiency and time-on-task measures can be used to build models for adaptive learning technologies, but the different strategies for computing efficiency and estimating time-on-task affect the accuracy of how learning is measured [199]—especially for programming tasks [214]. Furthermore, learners often engage in activities that are not related for learning during a learning task and researchers account for these gaps differently [48, 176, 199, 317].

Efficiency in the context of learning and instruction is defined as “the ability to reach established learning or instructional goals with a minimal expenditure of time, effort, or cognitive resources” and is dependent on the learner [161, p. 1]. Empirical studies of efficiency help us understand what influences the rate, amount, and quality of learning [161, 296, 355]. These studies tell us how much “time and effort are needed to master academic competencies” such as basic introductory computer programming concepts when solving different types of computer programming problems [161, p. 1] and how to improve cognitive learning gains. But regardless of the amount of *time* or *effort* spent acquiring new knowledge, researchers posit “improving the amount or quality of knowledge acquisition may be especially beneficial for school improvement initiatives designed to help low-achieving students and for students with disabilities who typically require more time to master important information” [161, p. 2].

With regard to cognitive learning outcomes, studies have provided evidence it is significantly more efficient to solve Parsons problems with adaptation than to solve equivalent write-code problems [339, 425] and equally as effective for pre-posttest learning gains [100], but there are several ways to measure efficiency.

2.3.1 Measurement

Educational psychologists have historically used two popular computational models to measure efficiency: the deviation model and the likelihood model [161]. The deviation model has been used to measure the discrepancy (effect size) between standardized scores (z scores) of learners' performance on a task (*P*) and the effort they invest in a task (*R*) to calculate efficiency (*E*) [272]:

$$E = (zP - zR)/\sqrt{2} \tag{2.1}$$

The likelihood model has been used to compute the ratio of output to input, where output is performance (*P*) and input is effort (*R*) or time (*T*) [161]:

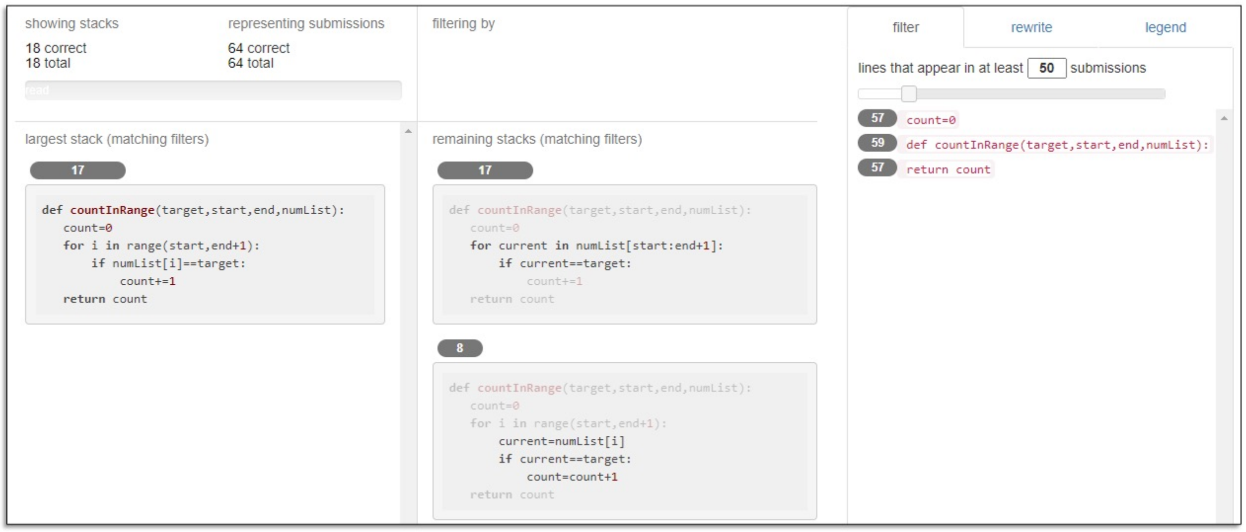
$$E = P/R \text{ or } P/T \tag{2.2}$$

Efficiency and problem-solving studies that utilize these models show mixed results and there is confusion about how to conceptualize efficiency [159]. Kalyuga and Sweller [181] calculated efficiency (via the likelihood model) to monitor student learning and tailor instruction in real-time based on changes in students' level of expertise; this proved reliable and resulted in higher knowledge and efficiency gains than instruction that did not adapt to the learner. Yet, in a recent study, Hoffman [159] compared efficiency on problem-solving tasks computed using both the deviation and likelihood models and revealed "fundamental differences in both the computational and measurement properties of the deviation and likelihood formulas, suggesting that both what is measured by each model, and how the constructs are measured, are indeed different" [p. 141]. Within educational psychology, conceptual confusion [see 41] about efficiency persists and researchers suggest using multiple measures [159].

Human-Computer Interaction (HCI) researchers define efficiency as "(1) the accuracy and completeness with which users achieve certain goals and (2) the resources expended in achieving them" [119]. In contrast to educational psychologist, HCI researchers consider efficiency to be an independent aspect of usability along with effectiveness ("the quality of a programming solution and error rates") and satisfaction ("positive attitudes towards the use of a system") [119, p. 345]. Efficiency, in this context, is measured by computing task completion and/or learning time, but HCI researchers warn that "for complex tasks [such as computer programming], efficiency measures are useless as indicators of usability unless effectiveness is controlled" [119, p. 345]. Effectiveness here is defined as the quality of a solution assessed on a five-point scale from 1 *"very low—failure, a completely wrong answer"* to 5 *"very high—brilliant answer."*

In this dissertation, efficiency (or time-on-task) for each problem is measured by computing the time to the first correct solution. The estimation heuristic to calculate 'spending time' involved removing any difference greater than five minutes between the last recorded timestamp and the current timestamp for a problem and time spent on other problems [51]. Five minutes was chosen because no interaction for more than five minutes likely meant that the student took a break, especially since the largest median time to solve any of the problems was less than ten minutes. Effectiveness is controlled for by only including in our analyses task completion times for solutions that were 100% correct. Adaptive Parsons problems were 100% correct if all the blocks were in the right order, had the right level of indentation for each block, and did not include distractor blocks. Write-code problems were 100% correct if they passed all of the unit tests. In these studies the effectiveness of write-code solutions was not assessed using scales like the five-point scale above because we used students' final correct solutions regardless of how many attempts they made. The commonality of written student solutions was analyzed using OverCode as shown in Figure 2.2. OverCode is a value-free visualization system that clusters similar solutions to programming problems using both static and dynamic analyses [127].

Researchers recommend continued investigation into the effectiveness of Parsons problems



18 correct
18 total

64 correct
64 total

lines that appear in at least 50 submissions

17

```
def countInRange(target,start,end,numList):
    count=0
    for i in range(start,end+1):
        if numList[i]==target:
            count+=1
    return count
```

17

```
def countInRange(target,start,end,numList):
    count=0
    for current in numList[start:end+1]:
        if current==target:
            count+=1
    return count
```

8

```
def countInRange(target,start,end,numList):
    count=0
    for i in range(start,end+1):
        current=numList[i]
        if current==target:
            count=count+1
    return count
```

```
57 count=0
59 def countInRange(target,start,end,numList):
57 return count
```

Figure 2.2: *OverCode visualization of solutions for those who solved Problem 2 in Table 5.6 as write-code problem first. The top-left panel displays the number of clusters (18), called stacks, and the total number of visualized solutions (64). The panel below this in the first column shows the largest stack which is comprised of 17 solutions. The second column displays the remaining stacks. The third column displays the lines of code occurring in the cleaned solutions of the stacks together with their frequencies [127].*

[89]. In chapters two through five, this dissertation expands the study of problem-solving efficiency for solving adaptive Parsons problems versus solving equivalent write-code problems to uncontrolled settings and tests hypothesizes about the commonality of Parsons problems solutions. It contributes to our understanding of the use of adaptive Parsons problems during lecture and the generation of Parsons problems to support pattern comprehension and pattern application.

2.4 Cognitive Load

2.4.1 Short- and Long-Term Memory Capacity

Learning is the process of storing new information in memory. Scientists have partitioned memory into sensory memory, short-term memory, and long-term memory, which range from milliseconds to years in duration respectively [123]. Short-term memory (or working memory) is the only type with a limited capacity (7 ± 2 items)—particularly for novel information [123]. Short-term memory functions to keep information in an active state and retrieve information from long-term memory during a learning task [332]. Strain is the result of trying to process new information beyond the capacity of short-term memory. Cognitive Load Theory (CLT) explains the types of strain learners encounter [373]. It describes the amount of information that working memory can hold and/or manipulate at one time. Cognitive load is a multifaceted construct that impacts complex learning tasks such as computer programming (see Figure 2.3). Computing education researchers (CEdRs) and instructional design researchers use knowledge of human cognitive architecture to optimize cognitive load and design instruction and assessment that aid learners in the efficient formation of accurate mental models and the development of computational thinking skills [77, 239, 312, 387]. In relation to efficiency and time-on-task, the objective of designing and developing instruction and assessments with cognitive cost (i.e., cognitive load imposed, cognitive resources invested, or mental effort spent) in mind is to support a return on investment concerning efficiency—"learning faster and without mental stress" [180, p. 388].

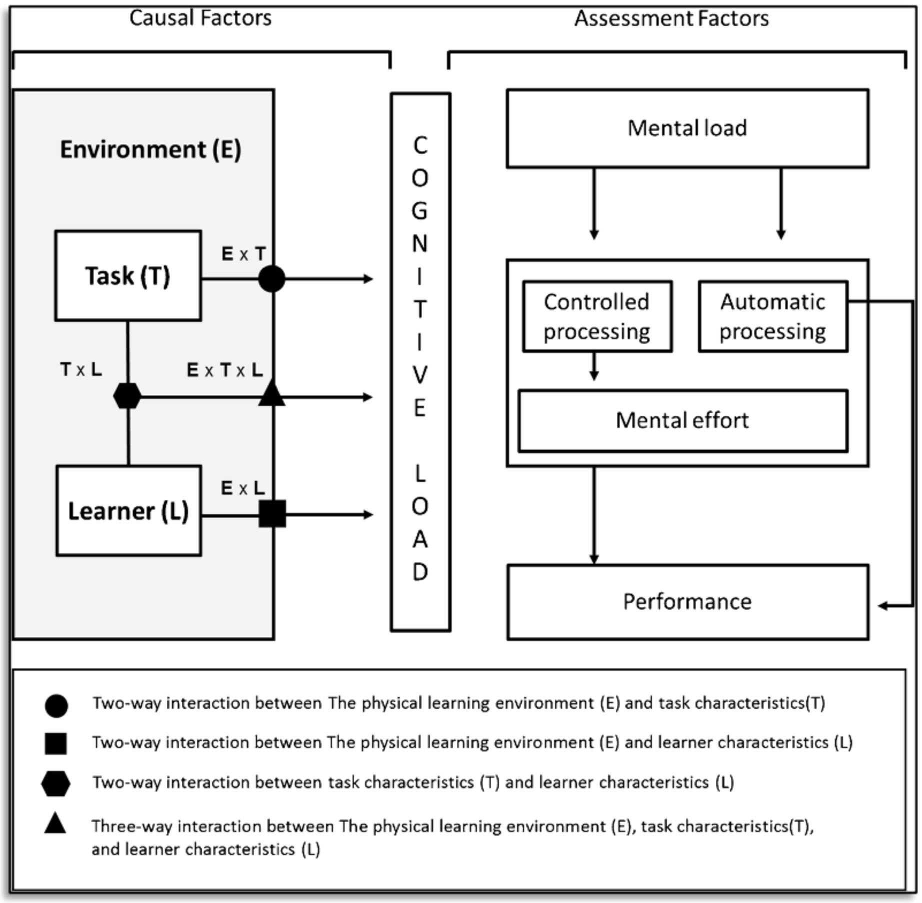

Causal Factors
Assessment Factors
Environment (E)
Task (T)
Learner (L)
E x T
T x L
E x T x L
E x L
COGNITIVE LOAD
Mental load
Controlled processing
Automatic processing
Mental effort
Performance
Two-way interaction between The physical learning environment (E) and task characteristics(T)
Two-way interaction between The physical learning environment (E) and learner characteristics (L)
Two-way interaction between task characteristics (T) and learner characteristics (L)
Three-way interaction between The physical learning environment (E), task characteristics(T), and learner characteristics (L)

Figure 2.3: *Choi, Van Merriënboer, and Paas' The construct of cognitive load. E = the physical learning environment, T = the learning task, L = the learner. Adapted and revised from Paas and VanMerriënboer (1994a), Educational Psychology Review, 6, p. 3. © Plenum Publishing Corporation. [63]*

2.4.2 Categories

Educational psychologists have divided this construct into intrinsic cognitive load, extraneous cognitive load, and germane cognitive load; however, the latter is up for debate [270].

Intrinsic cognitive load refers to the complexity of the information to be learned [373]. For example, novice programmers may have a harder time using nested structures versus flat structures because nested structures have more element interactivity and may require higher levels of abstraction [3, 4]. Extraneous cognitive load refers to the complexity of how the information to be learned is presented [373]. This load can be imposed by not adhering to the principles of coherence, redundancy, signaling, temporal contiguity, and spatial contiguity [240]. Extraneous cognitive load may be increased by ambiguous instructions and cluttered displays. Germane cognitive load refers to the effort required to learn during the learning task [373]. It can be imposed by tasks such as identifying subgoals or solving Parsons problems with distractors, but these tasks can lead to better learning outcomes [237, 254].

Each of these relates to element interactivity in a different way. Element interactivity refers to the complexity of learning new concepts that rely on having amassed prior knowledge of other concepts [58]. Intrinsic cognitive load is dependent on element interactivity [270]. Extraneous cognitive load is caused by nonessential element interactivity [270]. And germane cognitive load aids in dealing with element interactivity [270].

Other terms used to describe categories of cognitive load include mental load and mental effort. "Mental load is imposed by the task or environmental demands" and "mental effort refers to the amount of capacity or resources that is actually allocated to accommodate the task demands" [273, p. 354].

Regarding task difficulty and cognitive load, researchers posit "A difficult task does not necessarily lead to greater mental effort. Unless the individual is interested in the task and wants to do it well, task difficulty would lead the individual to give it up instead of trying to invest further effort. Thus conceptually, a task with high cognitive load should be a task that is perceived to be difficult and cannot be done well even though the individual likes the task and invests a high level of effort in it" [420].

2.4.3 Measurement

Empirical evidence of cognitive load can be collected indirectly, directly, subjectively, and through dual-task performance measures [191, 312]. Scientists have developed scales that are both unitary (combining categories) and deferential (subscales for each category) [192, 370].

The most valid and reliable measure is the Paas scale [271]. Computing education researchers have developed a differential scale from the Cognitive Load Component Survey (CLCS) [253], but it has provided mixed results [146, 253, 424]. Other validated measures of cognitive load include: the NASA Task Load Index (NASA-TLX) [147], the SOS scale [108, 366], and the differentiated cognitive load measure developed by Klepsch, Schmitz, & Seufert [191].

In this dissertation, the Paas scale is used to (1) address the fact that computing education researchers have been critiqued for not using validated subjective measures to survey the characteristics and attitudes of learners [281, 423], and (2) because prior research in computing education shows the Paas scale useful for understanding the mental effort imposed on novice programmers by the presentation of material [145, 146]. For example, Harms et al. found that when solving programming puzzles that are difficult novice programmers experience higher mental effort than working through tutorials for identical problems [145]. The Paas scale is valid, reliable, and easier to administer at scale than physiological techniques [374]. This last point was important because the scale itself and the frequency of administration can impose cognitive load.

The Computer Science Cognitive Load Component Survey (CS CLCS)—a three-component measure—is also used in this dissertation to explore why prior research with this scale has had mixed results [146, 253, 424].

2.4.4 Learning Tasks

Cognitive Load Theory (CLT) explains how we can design learning tasks to *reduce* intrinsic and extraneous cognitive load [392, 368]. This can be done by manipulating the complexity or the element interactivity of the information to be learned, clarifying instructional procedures, and designing the learning interface with human factors and cognitive ergonomics in mind [120, 204]—all of which are mediated by learner characteristics such as prior knowledge and ability [392]. Learning tasks that have to do with programming "generally involve working with a lot of information relating to the current state of the data represented and the processes being executed, the overall design and goals of the program, and also the language and tools being used" [312, p. 263]. Tasks like programming, which involve high element interactivity, result in high cognitive load for novice programmers because they lack the schemata necessary to free up working memory [311].

Completion problems provide novice programmers with partial solutions and are one way to reduce cognitive load. They are recommended for introductory computer programming [391]. Sweller, van Merriënboer, and Paas describe them as *"a bridge between worked examples and conventional problems"* [373, p. 268] because learners are forced to mindfully engage with the partial solutions. Adaptive Parsons problems are a type of completion problem. Similar to worked examples used to replace conventional forms of practice, adaptive Parsons problems are aimed at improving efficiency for learning how to program.

This dissertation explores the impact of using an interactive eBook with adaptive Parsons problems on cognitive load, the relationship between problem-solving efficiency and cognitive load, and order effects (i.e., how the order in which participants' solve each problem type relates to their mental effort).

2.5 Self-Efficacy

Robins [311, p. 25] states that "affective factors such as motivation, constructive attitudes to learning, positive expectations, and high self-efficacy or effort are also usually found to be correlated with success in programming."

Self-efficacy is defined as a person's belief "in their ability to influence events that affect their lives" [14, p. 1]. Educational researchers study learners' self-efficacy because it affects career choice and the development of competencies, values, and interests in selected environments. It is also correlated with academic achievement and persistence [14, 137, 274]. Inaccurate self-appraisal negatively affects performance and motivation [331]. Learners with negative self-efficacy beliefs for complex or difficult tasks in a given domain, such as computer programming, for example, are less resilient than learners with high self-efficacy beliefs who persevere [14]. And, Bandura himself stated that self-efficacy beliefs determine the amount of effort people expend on an endeavor [14].

Several researchers have studied self-efficacy in introductory computer programming courses [232, 423]. Ramalingam et al. [299] investigated the relationship between computer programming self-efficacy and mental models (a person's cognitive reproduction of real world objects and systems); they found mental models influence self-efficacy and that both programming self-efficacy and mental models influence academic achievement. Mental models (or schemas) are defined as the organization of information about how components of a system work into central concepts or topics [266]. They play an important role in the study of program comprehension/knowledge [50, 357]. In contrast, the study of programming strategies is concerned with how programming knowledge is used and applied (i.e., what are the common and uncommon strategies employed by novice programmers when solving programming problems and how can we increase knowledge/strategy transfer) [73, 309].

Other computing education researchers have found repeated failures, especially when first learning how to program, can lower students' self-efficacy [188]. Peer instruction increases self-efficacy [427]. Gender mediates self-efficacy beliefs (the relationship between females' self-efficacy and programming performance plateaus earlier in a CS1 course) [220], and gender also mediates the relationship between self-efficacy and a sense of belonging [32]. There is also a positive relationship between self-efficacy, a fixed mindset for programming, effort, and course grade [378]. Self-efficacy is significantly impacted by misconceptions about programming functions

[179]. And low self-efficacy is related to frequent negative self-assessments [130].

Few studies investigate self-efficacy, cognitive load (mental effort), help-seeking, and cognitive ability concurrently [see 382]. Yet, there is evidence that these relationships exist in related domains to computer science such as mathematics [160, 395]. Hoffman and Spatariu tested the motivational efficiency hypothesis and found beliefs about self-efficacy predict problem-solving efficiency (i.e. the ratio of problems solved correctly to time) and that "self-efficacy was a statistically significant predictor of problem-solving time" [p. 885 162]. And, furthermore, researchers suggest that in computer science learning, learners' self-efficacy needs to be accounted for to increase the adaptability of the systems they use [409].

To measure computer programming self-efficacy, researchers have developed the Motivated Strategies for Learning Questionnaire (MSLQ) [289], the Computer Programming Self-Efficacy Scale (CPSES) [300], and, most recently, the Introductory Programming Self-Efficacy Scale (IPSES) [361].

In this dissertation, the Introductory Programming Self-Efficacy Scale (IPSES) and self-efficacy portion of the Motivated Strategies for Learning Questionnaire (MSLQ) are used to explore the relationship between the problem-solving efficiency of completing adaptive Parsons problems and self-efficacy. The IPSES was used because its factorization is reported as more robust [361]. In chapter five, the scale's reliability is explored and the scores are used to cluster students into four groups based on prior research [188, 299]. In chapter six, the MSLQ was used to account for the frequency of asking learners about their self-efficacy; it consists of five items and learners were asked to fill it out once per week for eight weeks along with other instruments (see section 5.2.2 Materials).

2.6 Neurodiversity

Neurodiversity refers to individual variation in cognitive function, behavioral traits, affect, and sensory functioning differing from the general or 'neurotypical' population [318]. Judy Singer, a sociologist with Asperger syndrome and autistic rights advocate, coined this term in 1999 [349]. Proponents view autism, attention deficit hyperactivity disorder, Tourettes, dyslexia, hearing voices, bipolar disorder, down syndrome, dementia, and other neurominority experiences "as components on a broader continuum of sensory, affectual, and cognitive processing" [318, p. 2]. Neurodiversity is a challenge to the deficit (medical) model that portrays neurominorities as "ill, broken, and in need of fixing" where neurological deficits/disorders are exclusive to the individual [307, p. 1]. In contrast, the social model of disability is concerned with external forces that enforce restrictions on disabled people [318].

Most research on learning design has focused on neurotypical individuals [226] and there is relatively little research in computing education on learners with cognitive disabilities [198]. De Araújo and Andrade's systematic literature review on teaching programming to learners with cognitive disabilities showed that block-based programming and robotics were the most popular approaches to (1) teaching basic introductory programming concepts and (2) increasing inclusion and equity through appealing activities respectively [76].

2.6.1 Universal Design for Learning and Students with Disabilities in CS

eBooks that do not incorporate good instructional design principles run the risk of excluding some of their users. One pedagogical approach applied to teaching computer science education to learners with and without disabilities in K-12 and higher education is Universal Design for Learning (UDL) [5, 170, 171]. This framework was designed by the Center for Applied Special Technology (CAST) to increase access and engagement for the widest range of learners. Three broad principles founded on cognitive neuroscience research contribute to UDL [116]: (a) providing multiple means of engagement for learners to interact with learning materials (e.g. providing both Parsons problems and write-code problems), (b) providing multiple means of representation for comprehension (e.g. embedding the option to download eBook chapters for offline reading), and (c) providing multiple ways for learners to act out and express their understanding (e.g. creating discussion forums and facilitating peer instruction) (see https://udlguidelines.cast.org/).

Israel and Lash [171] posit two of the fundamental concepts that underlie UDL help to maximize students' strengths and reduce instructional barriers, although accessibility barriers and cognitive load barriers can pose a challenge. The first principle is that "Learner variability is the norm, not the exception" [171, p. 219]. We each learn in different ways and, although it is essential to consider social markers such as race, gender, dis/ability, sexuality, etc. to address learners' individual needs, "when we look at students only within categories (e.g., gifted, learning disability), we oversimplify differences between learners and do not fully acknowledge the diversity among them" [171, p. 219]. The second principle is that "Disability is contextual" [171, p. 219]. Instructional materials and learning technologies can disable students' learning at any time due to accessibility barriers and challenges. For example, if an introductory computer programming course course required students to use an integrated development environment (IDE) without an onboarding process this might limit their ability to succeed in the beginning of the course.

UDL can help learners with and without disabilities in postsecondary education overcome challenges and barriers related to learning and socialization when using computers [5, 330]. It can help improve equity and access for all learners because it supports the dispersion and neurodiversity that characterizes cognition and learning [5, 12]. Nonetheless, most research has focused on K-12 learners [302]. In this dissertation, I focus on the design of an eBook with regard to UDL principles for neuordiverse learners in tertiary education. This learning took place outside of the classroom—informally. The goal was to investigate the accessibility of adaptive Parsons problems and equivalent write-code problems for programming practice.

2.6.2 Accessibility of Adaptive Parsons Problems

The majority of block-based programming research on accessibility has focused on programmers with visual or motor impairments [247, 248]. Several challenges exists for learners with visual impairments including: navigating code, comprehending code, debugging code, and skimming code [258]. And studies have mainly focused on K-12 learners with visual impairments and block-based environments that use the Scratch visual programming language [257]. To date, no one has explored the accessibility of adaptive Parsons Problems. Moreover, most of the studies on adaptive Parsons problems have focused on what neurotypical individuals prefer (adaptive Parsons problems or write-code problems) and their computational practices, perspectives, and attitudes (how they solve adaptive Parsons problems, whether they find them useful, and if they comprehend intra- and inter-problem adaptation) *inside* the classroom [104, 102, 100, 98]. Hence, this dissertation also explores the accessibility of adaptive Parsons Problems for neurodiverse learners and their computational practices, perspectives, and attitudes while using an interactive eBook to learn how to program *outside* of the classroom.

CHAPTER 3

Parsons Problems as Active Learning Lecture Activities

Adaptive Parsons problems could be used to reduce the difficulty of introductory programming courses and increase the use of active learning in lecture. Parsons problems provide mixed-up code blocks that must be placed in order. If a learner is struggling to solve an adaptive Parsons problem, it can dynamically be made easier. This makes it possible for students to correctly solve a problem in a limited amount of time, even if they are struggling. Previous research on the effectiveness and efficiency of solving Parsons problems for learning has been conducted in controlled conditions in lab or during discussion. During lecture, we assigned undergraduates adaptive Parsons and equivalent write-code problems to solve. We tested the efficiency of solving each problem type through three between-subjects experiments. The median time to solve each Parsons problem was less than the median time to write the equivalent code for all but two of the problems—both requiring knowledge of complex conditionals. However, that difference was significant for only six of the ten problems. Our hypothesis for why the two problems had a higher median time to solve as an adaptive Parsons problem than as a write-code problem was that the problem instructions did not match the Parsons problem solution and/or there were a large number of possible correct solutions. Results from student surveys also provided evidence that most students (78%) found solving adaptive Parsons problems in lecture as helpful for their learning, but that some (36.2%) would rather write the code themselves. These findings have implications for how to best use Parsons problems.[1]

3.1 Introduction

Introductory computing classes at the college level had an average failure rate of 28.3% in 2018 which does not seem particularly high compared to college algebra courses in the United States that

[1]Portions of this chapter were adapted from [97]

can reach 50% [27]. But most computing courses use traditional passive lecture and assignments that require students to write code from scratch [187, 286]. And only 55% of course syllabi explicitly state that writing programs is one of their learning outcomes/objectives which may impact students' ability to gauge how much effort is required to learn—especially first-generation students [20, 136]. Novice programmers often spend many frustrating hours trying to figure out why their program does not compile or does not produce the expected output [23]. Furthermore, students that encounter errors while programming experience negative emotions that impact self-efficacy [187]. High self-efficacy improves persistence in a field, while low self-efficacy increases the odds that students will fail or change majors [95]. Instead of traditional lecture, active learning techniques can improve learning and increase retention in computer science courses [47, 118, 138, 243].

Diversity is a persistent problem in computing. There continues to be a "leaky pipeline" for students from underrepresented groups who tend to have less prior experience [49, 235, 234, 398]. A survey of undergraduate students who had dropped an introductory computing course found that nearly half thought it was *too* challenging and 75% of those were female [112]. Negative experiences in courses tend to affect women more than men [93, 235] which may be one reason why women are more likely to leave the CS than men, even if they have better grades than the men who stay [183]. In 2020, female students earned only 20.6% of computer science bachelor's degrees, 16.6% of computer engineering, and 29.4% in information; nonbinary/other students earned 0.1%, 0%, and 0% respectively [430]. Bachelor degrees earned by underrepresented minorities (URMs) were portioned into 8.4% for Black/African American students, 11% for Hispanic/Latino students, 0.3% for American Indian and/or Alaska Native students, and 0.2% for Pacific Islander students [169]. Gender-focused efforts have failed to increase the percentage of Black women in computing [301]. For students with disabilities, enrollment in undergraduate programs "does not translate into a strong pipeline" [343, p. 1]. And due to low sense of belonging, LGBTQIA+ students often leave computing degree programs [363]. Active learning techniques can improve motivation and learning for students from underrepresented groups [195, 380].

In this study, we explore using Parsons problems as active lecture assignments. The cost of learning can be high for complex tasks and students learn at different rates [35, 51]. Writing a program from scratch is appealing in that it is seen as authentic but it can be time-intensive [338]. However, while writing a complete program novices may experience high cognitive load, which can quickly overwhelm them and actually impede learning and decrease motivation [390, 392]. One of the recommended ways to reduce cognitive load during programming is to have students complete a partial program rather than write a complete program [389]. Parsons problems are a type of code completion problem where the learner must place mixed-up code blocks in order [100]. Students from underrepresented groups perform better on these types of block-based problems [185, 407]. Two controlled experiments have provided evidence that solving Parsons problems (both non-adaptive and adaptive) are more efficient, but just as effective for learning, than writing the equivalent code [102, 100]. These experiments were conducted in a closed lab with undergraduate volunteers. While lab-based experiments can provide evidence for the effectiveness of new approaches, it is also important to validate these findings in real educational contexts [16]. In addition, prior research provided evidence that secondary teachers perceived that solving adaptive Parsons problems helped them learn to fix and write code [98]. However, it is important to also determine undergraduate students' attitudes towards solving adaptive Parsons problems in lecture. One reason instructors do not adopt new teaching approaches is that they fear students will not like them [17, 40].

In this study the following research questions were put forth:

RQ1: What is the effect on efficiency (time to correct solution) of solving adaptive Parsons problems with distractors versus writing the equivalent code when used as lecture assignments?

RQ2: What are undergraduate students' attitudes towards adaptive Parsons problems as lecture assignments?

3.2 Methods

We used both a between-subjects experiment as well as an end-of-course student survey to answer the research questions.

3.2.1 Context

This research was conducted at a the University of Michigan School of Information. All participants were enrolled in *Data-Oriented Programming*. We received IRB permission to analyze anonymous clickstream data from the course.

This course is the second required Python course for School of Information majors, although other majors take it as well. Undergraduates typically take it in their first or second semester. It requires prior programming experience (variables, loops, conditionals, and functions). The course focuses on developing intermediate programming skills in Python and covers working with data from a variety of sources (strings, files, APIs, websites, and databases), object-oriented programming basics, regular expressions, debugging, testing, and SQL.

Lecture was an 80 minute period that met twice a week in an auditorium-style lecture room. Lecture switched to online during March of 2020 due to Covid. The instructor lectured for 10 to 15 minutes at a time and then asked a peer instruction question [71]. Peer instruction has been known to substantially reduced failure rates [291], improved retention [292], and increased final exam performance [347].

One of the researchers created practice problems in a free eBook for most lectures. The practice problems were typical of those used in a first course for computing majors. During lecture the instructor assigned a set of problems for the students to work on either individually or with a partner. Most students worked individually. Students were given 20 minutes during lecture to work on an assignment, but had a week to complete it in order to allow students who were sick or had more difficulty with the assignment to complete it. Students earned one point for correctly completing each problem. Each assignment typically had five problems. Students could earn up to a total of 200 points for doing the lecture assignments and could drop several of the assignments.

Students also attended a 50 minute discussion section once a week in smaller groups of about 20 to 25 students. In discussion, students wrote code in small groups to solve problems that were similar to those on the homework or projects. Students also had to write code for the nine homework assignments and three projects. Projects were expected to be about twice as difficult as homework assignments.

An end-of-course student survey is always administered in the final weeks of the course. The instructor added three questions to this survey in the fall of 2019 and the winter of 2020. The instructor also added a couple of open-ended questions during the winter of 2020.

3.3 Between-Subjects Experiment

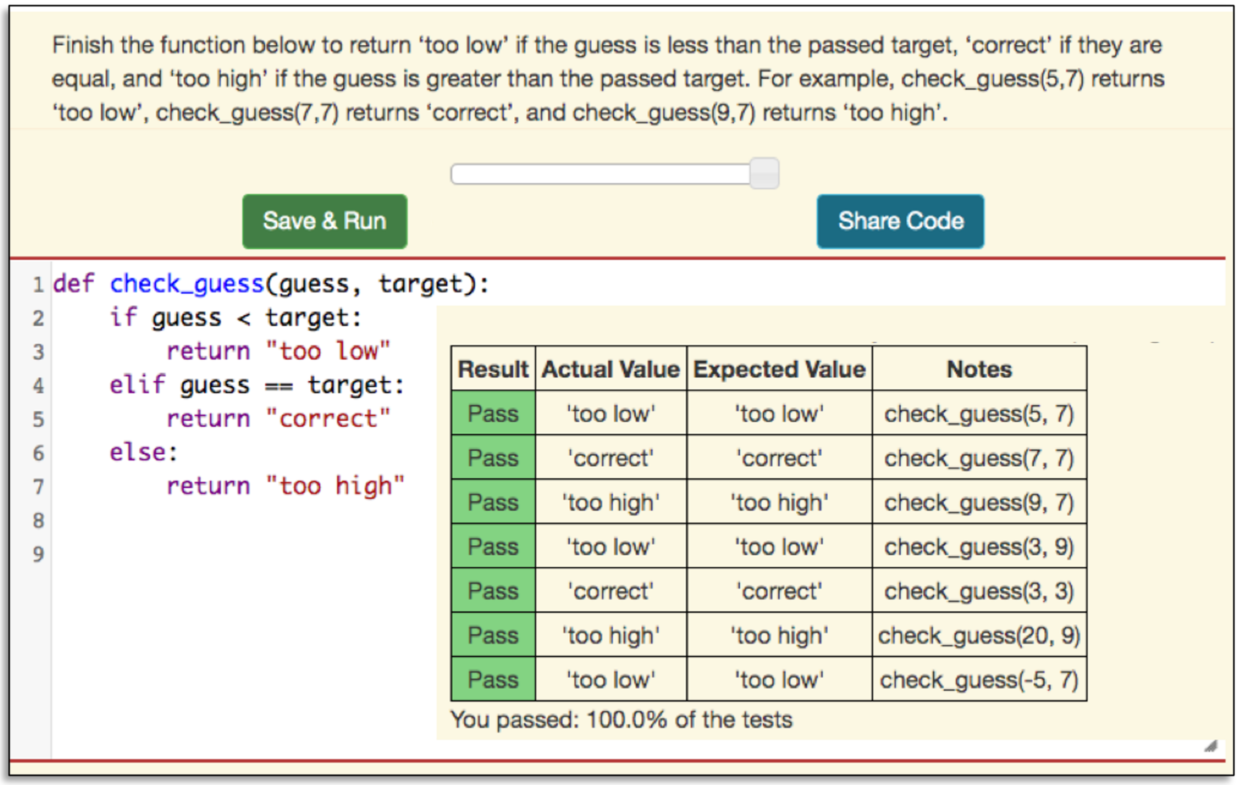
We conducted three between-subjects experiments in lecture during the Winter of 2020. Forty-nine percent identified as female and the rest as male (*N* = 152); the ages ranged from 18 to 36 years old. Eleven percent identified as Black or Hispanic, 36% as Asian, 42% White, 8% as Multiracial, and 4% did not indicate their race. The average GPA was 3.433 (*SD* = 0.419).

3.3.1 Materials

We developed three assignments with five problems in each and each had two versions: A and B (see 3.1). The first two assignments both had four pairs of Parsons problems and write-code problems, while the third assignment only had two. In version A, the first problem was a write-code problem while in version B the same problem was a Parsons problem. See the first problem as a write-code problem with unit tests in Figure 3.1 and as a Parsons problem in Figure 3.2. The problem type alternated between write-code and Parsons for the first four problems in the first two assignments. The last problem was the same fix-code problem.

Table 3.1: Order and Type of Problem by Version

Version	Problem Type
A	Write, Parsons, Write, Parsons, Fix
B	Parsons, Write, Parsons, Write, Fix


Finish the function below to return 'too low' if the guess is less than the passed target, 'correct' if they are equal, and 'too high' if the guess is greater than the passed target. For example, check_guess(5,7) returns 'too low', check_guess(7,7) returns 'correct', and check_guess(9,7) returns 'too high'.

```python
def check_guess(guess, target):
    if guess < target:
        return "too low"
    elif guess == target:
        return "correct"
    else:
        return "too high"
```

Result	Actual Value	Expected Value	Notes
Pass	'too low'	'too low'	check_guess(5, 7)
Pass	'correct'	'correct'	check_guess(7, 7)
Pass	'too high'	'too high'	check_guess(9, 7)
Pass	'too low'	'too low'	check_guess(3, 9)
Pass	'correct'	'correct'	check_guess(3, 3)
Pass	'too high'	'too high'	check_guess(20, 9)
Pass	'too low'	'too low'	check_guess(-5, 7)

You passed: 100.0% of the tests

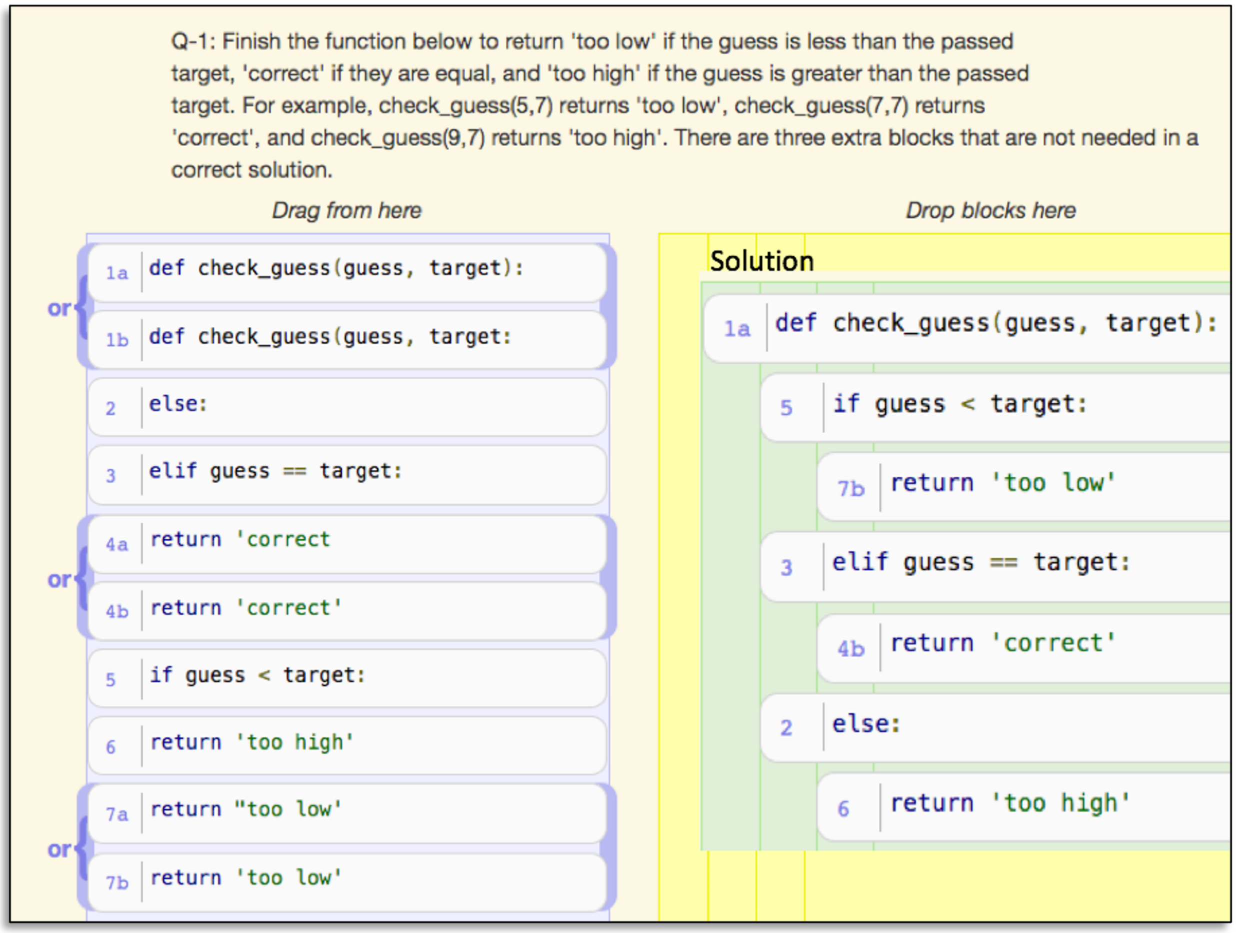
Figure 3.1: *Write-Code Problem (Problem One in Table 2).*

3.3.1.1 Experimental Protocols/Study Design

The first assignment was released in week one. It covered functions, complex conditionals, and strings. Students with an odd birth month were assigned to version A and those with an even birth month were assigned to version B.

The second assignment was released in week two. It covered functions, conditionals, strings, lists, and loops. Students who were seated on the left side of the lecture facing the screen completed version A while those on the right side did version B. An example write-code problem from this set (problem seven in Table 2) is shown in Figure 3.3.

The third problem assignment was administered in week four. It covered creating classes. This assignment had two problems that we tested as both a Parsons problem and write-code problem. You can see one of these problems in Figure 3.4. Students were asked to pick a random number


Q-1: Finish the function below to return 'too low' if the guess is less than the passed target, 'correct' if they are equal, and 'too high' if the guess is greater than the passed target. For example, check_guess(5,7) returns 'too low', check_guess(7,7) returns 'correct', and check_guess(9,7) returns 'too high'. There are three extra blocks that are not needed in a correct solution.

Drag from here

```
1a def check_guess(guess, target):
or
1b def check_guess(guess, target:
2 else:
3 elif guess == target:
4a return 'correct
or
4b return 'correct'
5 if guess < target:
6 return 'too high'
7a return "too low'
or
7b return 'too low'
```

Drop blocks here

Solution

```
1a def check_guess(guess, target):
    5 if guess < target:
        7b return 'too low'
    3 elif guess == target:
        4b return 'correct'
    2 else:
        6 return 'too high'
```

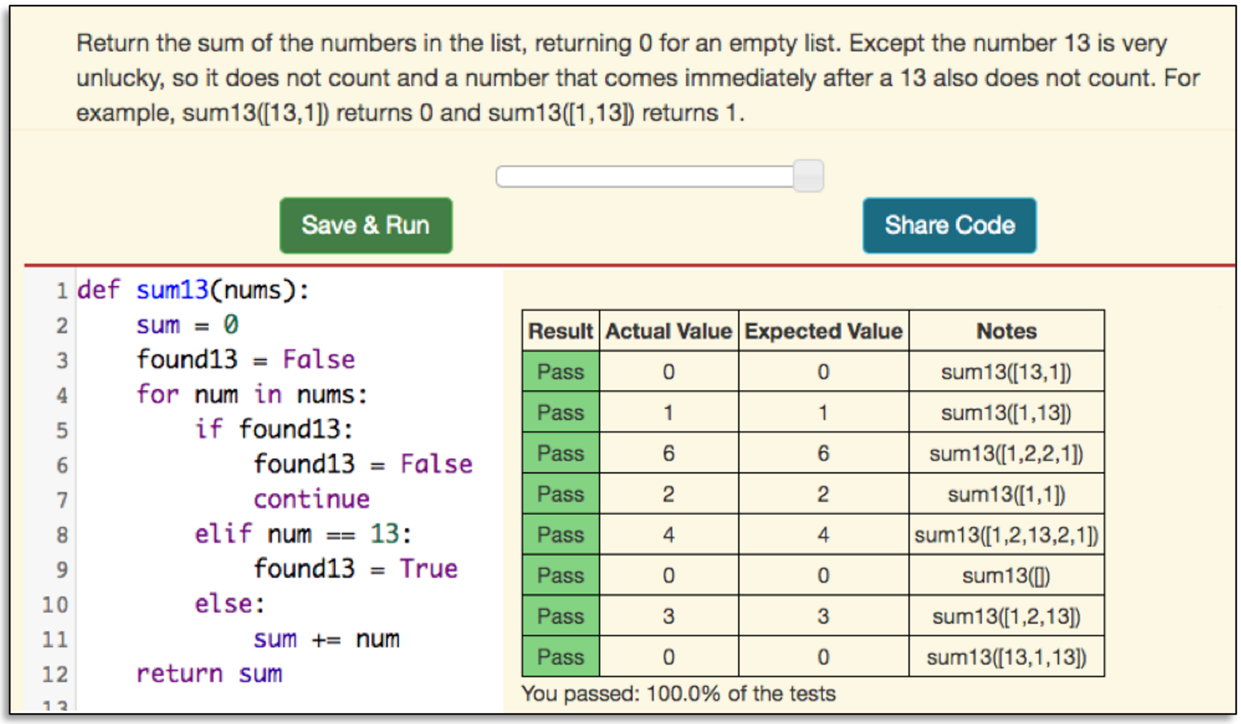
Figure 3.2: *Parsons Problem with Mixed-up Blocks on the Left and the Solution on the Right (Problem One in Table 2).*

from 1 to 10 and if the number was even they were to solve one version and if odd the other.

3.3.1.2 Analysis

We calculated the time in seconds to a correct solution. We removed any dead time (no interaction in the eBook for more than five minutes) and any time spent on solving other problems.

Since this was a between-subjects study and the data violated assumptions of normality and equal variances we ran Mann-Whitney U tests to analyze the difference in the times to the first correct solution between the groups [260]. We also report probability-based effect sizes as this measure is reported to be more robust when parametric assumptions are violated [66, 323].



Return the sum of the numbers in the list, returning 0 for an empty list. Except the number 13 is very unlucky, so it does not count and a number that comes immediately after a 13 also does not count. For example, sum13([13,1]) returns 0 and sum13([1,13]) returns 1.

```python
def sum13(nums):
    sum = 0
    found13 = False
    for num in nums:
        if found13:
            found13 = False
            continue
        elif num == 13:
            found13 = True
        else:
            sum += num
    return sum
```

Result	Actual Value	Expected Value	Notes
Pass	0	0	sum13([13,1])
Pass	1	1	sum13([1,13])
Pass	6	6	sum13([1,2,2,1])
Pass	2	2	sum13([1,1])
Pass	4	4	sum13([1,2,13,2,1])
Pass	0	0	sum13([])
Pass	3	3	sum13([1,2,13])
Pass	0	0	sum13([13,1,13])

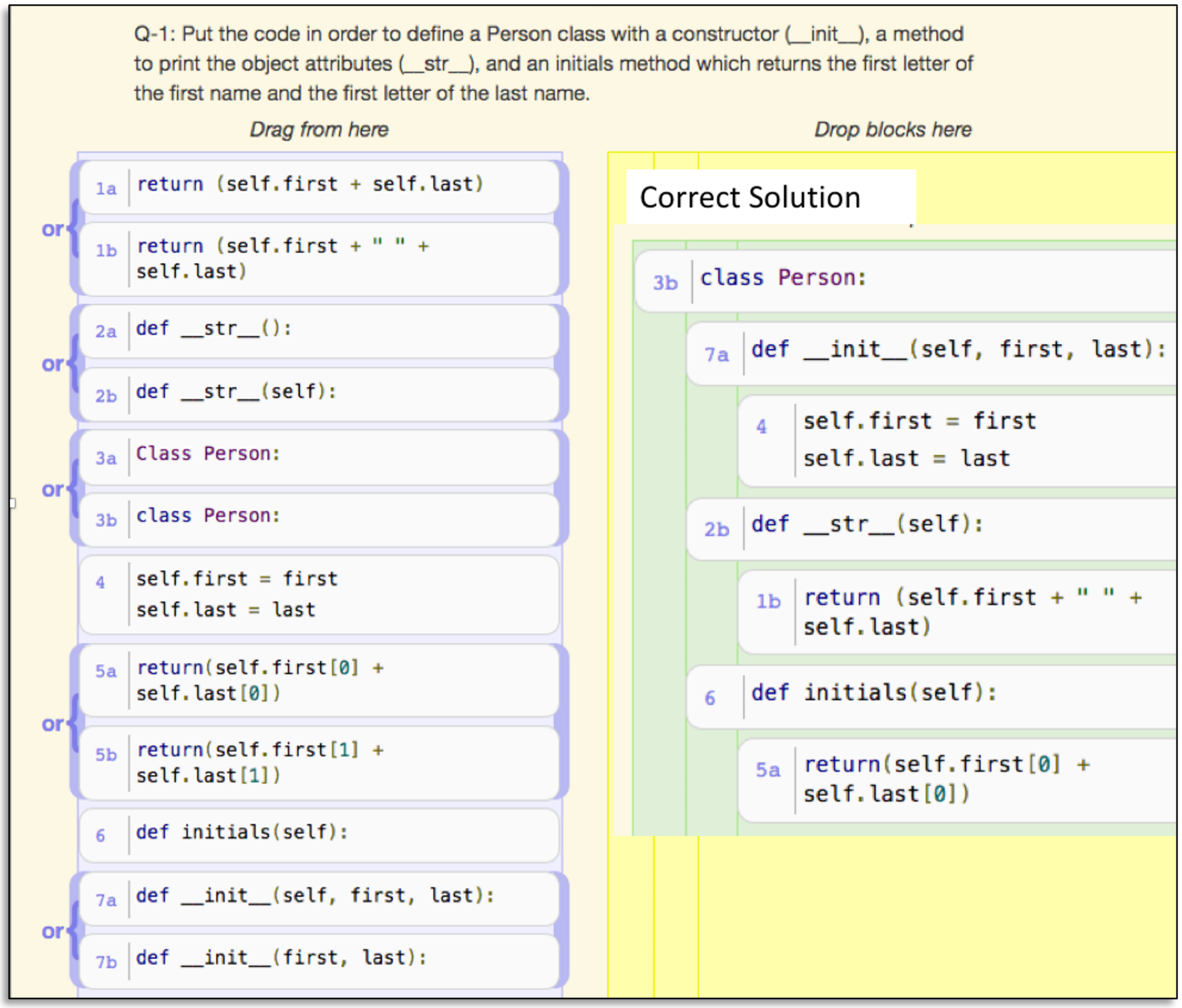
You passed: 100.0% of the tests

Figure 3.3: *Example Write-Code Problem from the Second Problem Set (Problem Seven in Table 2).*

3.3.2 Results

The median time to correctly complete each problem as a Parsons problem and as a write-code problem is shown in Table 3.2. Note that the median time to complete each Parsons problem was less than the median time to complete the equivalent write-code problem for eight of the ten problems. However, that difference was only significant for six of the problems. This result is different from prior research which found that Parsons problems were significantly faster to solve than writing the equivalent code, however that study was conducted in controlled conditions and all the Parsons problems were about loops, lists, iteration, and simple conditionals. The median time to complete the alarm clock and speeding problems shown in Figure 3.5 was actually larger for the Parsons problem than the write-code problem. This was significant for the alarm clock problem. Both of these problems were about complex conditionals.

The effect size (*A*) shown in Table 3.2 means that that a student randomly selected from the write-code group would probably take from 53% to 85% longer to correctly solve the same problem than a student randomly selected from the Parsons problem group.


Q-1: Put the code in order to define a Person class with a constructor (__init__), a method to print the object attributes (__str__), and an initials method which returns the first letter of the first name and the first letter of the last name.

```
1a  return (self.first + self.last)
1b  return (self.first + " " +
    self.last)
2a  def __str__():
2b  def __str__(self):
3a  Class Person:
3b  class Person:
4   self.first = first
    self.last = last
5a  return(self.first[0] +
    self.last[0])
5b  return(self.first[1] +
    self.last[1])
6   def initials(self):
7a  def __init__(self, first, last):
7b  def __init__(first, last):
```

Correct Solution

```
3b  class Person:
7a      def __init__(self, first, last):
4           self.first = first
            self.last = last
2b      def __str__(self):
1b          return (self.first + " " +
            self.last)
6       def initials(self):
5a          return(self.first[0] +
            self.last[0])
```

Figure 3.4: *Example Parsons Problem from the Third Assignment (Problem Nine in Table 2).*

3.4 Why Less Efficient at Solving Parsons Problem Three?

Since problem three (alarm clock) as shown in Figure 3.5 took students significantly longer to solve as a Parsons problem than as a write-code problem, we investigated it further. We examined the student written code. Of the 60 students who successfully wrote code to solve this problem, only 4 (6.7%) used a solution that was equivalent to the provided solution to the Parsons problem. A few students used just four complex conditionals as shown in Figure 3.6.

Most students solutions used nested conditionals. However, some first tested the day of the week and then the vacation flag. Some tested if vacation was false first rather than true. There were also many approaches to testing the day to see if it was a weekend or weekday including explicit checking of the weekend days, testing if the day was greater than zero and less than six,

Table 3.2: Time to Complete Parsons Problem vs. Write-Code Problem

	Parsons Problem		Write-Code Problem		Mann-Whitney U		
Problem	*N*	*Mdn* in seconds (*SD*)	*N*	*Mdn* in seconds (*SD*)	*U*	*p*-value	*A*
1 Check Guess	83	86.5 (79.15)	68	91 (129.81)	2648.5	0.71	0.53
2 Get Middle	66	319 (245.81)	64	372 (457.81)	1854	0.36	0.57
3 Alarm Clock	73	268 (582.21)	60	208 (214.28)	2570.5	0.04*	0.38
4 Speeding	64	231 (266.69)	65	178.5 (182.69)	2243	0.27	0.46
5 Loop Average	73	99.5 (77.7)	65	375.5 (288.91)	587	0.00***	0.85
6 Sum Odd	74	72 (96.37)	70	101 (204.03)	1650	0.00***	0.67
7 Sum 13	70	367 (282.53)	65	596.5 (749.37)	1806	0.07	0.65
8 Filter Words	72	63 (749.37)	69	104 (202.99)	1581	0.00***	0.69
9 Person Class	87	69.5 (76)	53	169.5 (213.28)	878	0.00***	0.73
10 Car Class	57	89 (78.23)	77	231.5 (257.92)	662.5	0.00***	0.76

Note. * p < .05, ** p < .01, *** p < .001. *A* = probability-based effect size measure (nonparametric generalization of common language effect size statistic).

using a range, checking if the day was in a list, and checking if the day modulo six was zero.

Since line-based feedback Parsons problems can only have one correct solution they may not be more efficient than writing the equivalent code when there are many possible correct solutions, such as in this case. In fact, we tested this problem again in a between-subjects study in the fall of 2020 after revising the instructions to try to lead students to the provided Parsons problem answer. In that study, 77 students solved it as a Parsons problem with a median time to solve of 229 seconds and 83 solved it as a write-code problem with a median time of 224 seconds. The difference was not significant using a Mann-Whitney U test. More work needs to be done to test if Parsons problems are best used with problems that do not have a large variety of possible solutions. Other research has provided evidence that Parsons problems may not be significantly faster to solve than writing the equivalent code when the Parsons solution is unusual [154]. It could be that Parsons problems are best used with problems that have a single most common solution.

3.5 End-of-Course Student Survey

It is important to gather student attitudes and feedback on new types of assignments. One reason that instructors give for not engaging in active learning during lecture is that they are concerned that they might receive negative student evaluations [17, 40]. The instructor added statements to the end of course student survey in both Fall 2019 and Winter 2020. Students answered using a five point Likert scale where one was "Strongly Disagree" and five was "Strongly Agree". All students would have received extra points if the response rate reached a specified level (ten points if 90% for Fall 2019 and five points if 85% for Winter 2020). However, the response rate did not reach the specified level either semester.

3.5.1 Student Survey Results from Fall 2019

The response rate was 83% (108 out of 130) for Fall 2019, however not every student answered every question. The first question was, *"I found mixed-up code problems in lecture practice helpful for learning"*. As you can see from Table 3.3, 78.3% of the respondents either agreed or strongly agreed. However, 11% disagreed or strongly disagreed that mixed-up problems in lecture were helpful for learning. This could be due to expertise reversal effect [371], in which techniques that help novice students can actually increase the cognitive load of more expert students. Prior research with Parsons problems has shown that experts can find solving a Parsons problem harder than writing the equivalent code, especially when the Parsons problem solution does not match the expert's solution [165].

Table 3.3: Student Responses on an End of Course Survey from Fall 2019 and Winter 2020

Term	Num	Question	N	1 (SD)	2 (D)	3 (N)	4 (A)	5 (SA)
Fall 2019	1	I found the mixed-up code problems in lecture practice helpful for learning.	106	0.9%	9.4%	11.3%	50.0%	28.3%
Fall 2019	2	I would rather write the code myself than solve a mixed-up code problem.	105	9.5%	29.5%	24.8%	21.0%	15.2%
Fall 2019	3	I would like to have the choice of solving the mixed-up code problem or writing the equivalent code.	101	1.0%	8.9%	34.7%	36.6%	18.8%
Winter 2020	1	I would rather write code from scratch than solve a mixed-up code problem.	105	9.5%	30.5%	29.5%	15.2%	15.2%
Winter 2020	2	I found the "Help Me" button on the mixed-up code problems useful.	103	1.0%	4.9%	13.6%	44.7%	35.9%

Note. SD = Strongly Disagree, D = Disagree, N = Neutral, A = Agree, and SA = Strongly Agree

The second question was, *"I would rather write the code myself than solve a mixed-up code problem."* While 36.2% of respondents agreed or strongly agreed with this statement 39.0% either disagreed or strongly disagreed. This provides evidence that while most students find solving Parsons problems useful, over a third of them would rather write the equivalent code.

The third statement was *"I would like to have the choice of solving the mixed-up code problem or writing the equivalent code."* Of the respondents, 55.4% agreed or strongly agreed with this statement, while 9.9% disagreed or strongly disagreed.

3.5.2 Student Survey Results from Winter 2020

In the Winter of 2020, we added statements about Parsons problems to the end-of-course-student survey. The survey response rate was 77.5% (107 out of 138), however not every student answered every question.

The first added statement was *"I would rather write code from scratch than solve a mixed-up code problem."* Of the respondents, 40% disagreed or strongly disagreed with this statement and 30.4% agreed or strongly agreed as seen in Table 3.3. It is interesting to note that the percentage who agreed or strongly agreed (30.4%) with this statement was less than that for a similar statement (number two) from Fall 2019 (36.2%), while the percentage that disagreed or strongly disagreed was very similar (about 40%).

The second statement was, *"I found the 'Help Me' button on the mixed-up code problems useful."* The Parsons problems have a "Help Me" button that triggers adaptation. Learners can click the "Help Me" button to dynamically make the current problem easier if they have submitted at least three incorrect solutions. Each time the learner clicks on the "Help Me" button it will remove a distractor block (if there are any), or provide indentation (if needed), or combine two blocks into one until there are only three blocks left. Of the respondents, 80.6% agreed or strongly agreed that the "Help Me" button was useful, while 5.9% disagreed. This provides evidence that most undergraduate students value adaptation.

3.6 Discussion

The median time to solve each Parsons problem was less than the median time to solve the equivalent write-code problem for eight of the ten problems studied, but that difference was only significant for five of those problems. This result differs from prior research in a controlled environment [102]. It may be that students in a controlled environment were more on task than students who were solving problems as part of lecture and who could complete the problems anytime over the next week.

Two problems, both with complex conditionals, actually had a higher median time to solve as Parsons problems than as a write-code problems. Our hypothesis is that the instructions for these problems did not match the Parsons solution and there are many different ways to correctly solve these problems. We revised the instructions for one of these problems (alarm clock) and found that this problem was not significantly faster to solve as a Parsons problem than as a write-code problem. Further work needs to be done to determine why this is the case, but it may be that Parsons problems are not significantly faster to solve when a problem has many possible solutions.

The end-of-course student surveys showed that while the majority of students (78.3%) found using Parsons problems in lecture helpful for learning, about a third of them would rather solve the equivalent write-code problem, and over half of them would like the choice. This is consistent with expertise reversal effect in which techniques to help novice students can actually increase the cognitive load for experts [371]. Parsons problems are a type of scaffolding to help students learn to write code. It is important that scaffolding fades as the learner develops expertise [141]. We have since added the ability for a student to switch to an equivalent write-code problem with unit tests when presented with a Parsons problem.

Most students (80.6%) found the intra-problem adaptation ("Help Me" button) useful, but about 6% did not. This suggests that more work should be done to make the adaptation process more understandable to learners. Perhaps we should provide an explanation about why a distractor is incorrect when it is removed from the solution during adaptation.

3.7 Limitations

This study was conducted in Python in one class at one university. More research should be done to test the results in other languages, classes, and contexts.

It is possible that there were significant differences between the groups in terms of prior experience, which could have affected the median completion time. However, students were randomly assigned to the groups using a different method for each assignment. That should mitigate this concern.

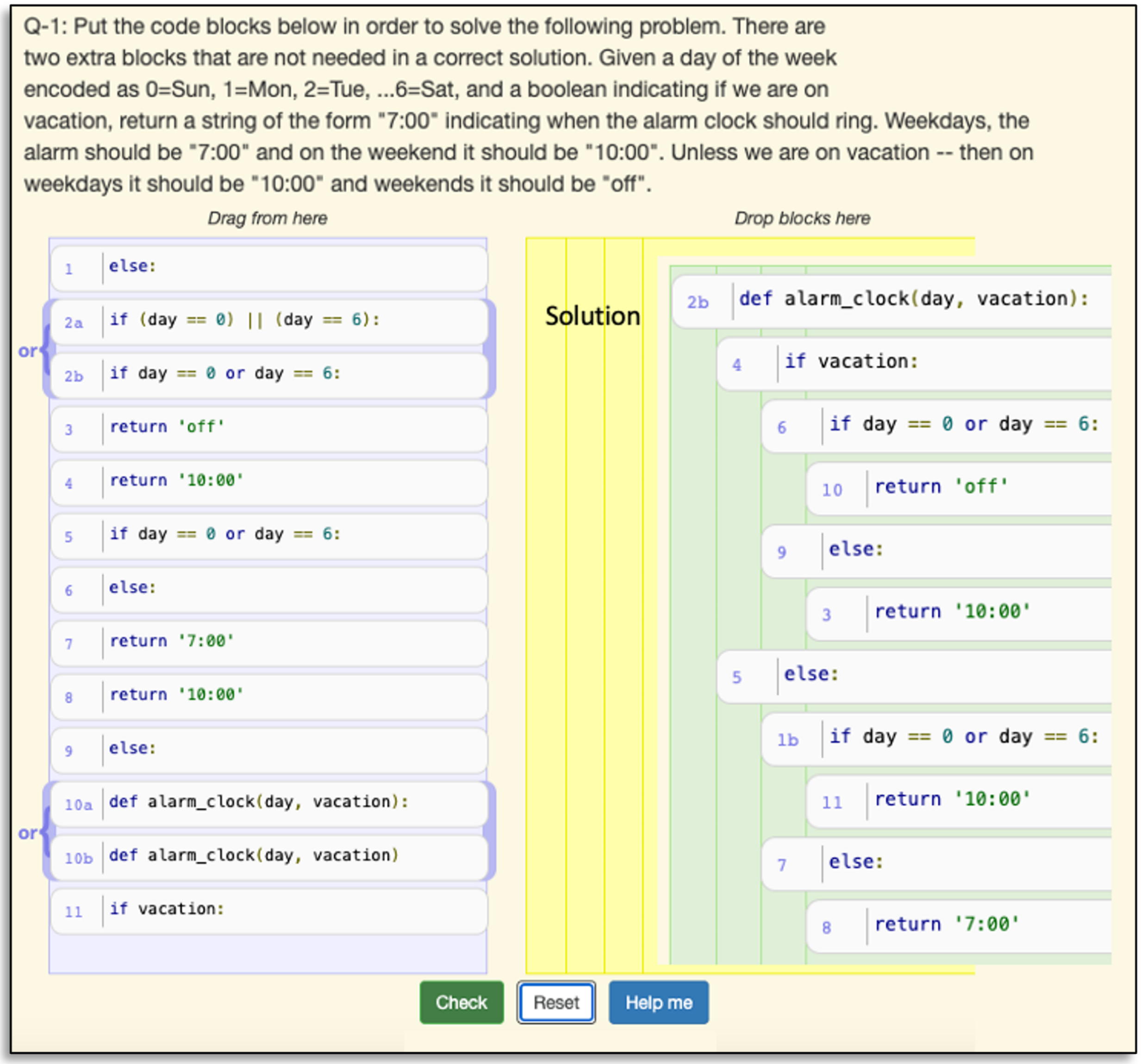
While the problems were given in lecture and students were given time to complete the problems in lecture, it is possible that they completed them after lecture since they were not actually due for a week. Both groups A and B were in this situation which should still make the data comparable. We took this situation into account when calculating the time to the first correct solution by removing any time spent solving other problems and any gaps in interaction of more than five minutes.

This study did not compare the learning gains from solving Parsons problems versus writing the equivalent code. However, prior research has shown no significant difference in learning gains [102, 100, 425]. Still, more research should test the learning gains from solving adaptive Parsons problems versus writing the equivalent code as active learning exercises in lecture.

3.8 Conclusion

This research provides evidence that learners can solve adaptive Parsons problems significantly more efficiently than writing the equivalent code as active learning exercises in lecture, but not always. The evidence suggests that a Parsons problem may not be more efficient to solve when there are many possible correct solutions, however more work should be done to verify this hypothesis. If this hypothesis is true, it has implications for the types of problems that should be used as Parsons problems.

This paper also provides evidence that most undergraduate learners find solving adaptive Parsons problems as active learning assignments in lecture useful for learning. Since active learning during lecture can improve student learning, motivation, and retention, we suggest that more computing instructors try adding adaptive Parsons problems assignments during lecture. Adaptation can help even struggling students successfully complete a Parsons problem, which makes this type of problem ideal for the lecture environment where it is difficult to provide individual help to struggling students.

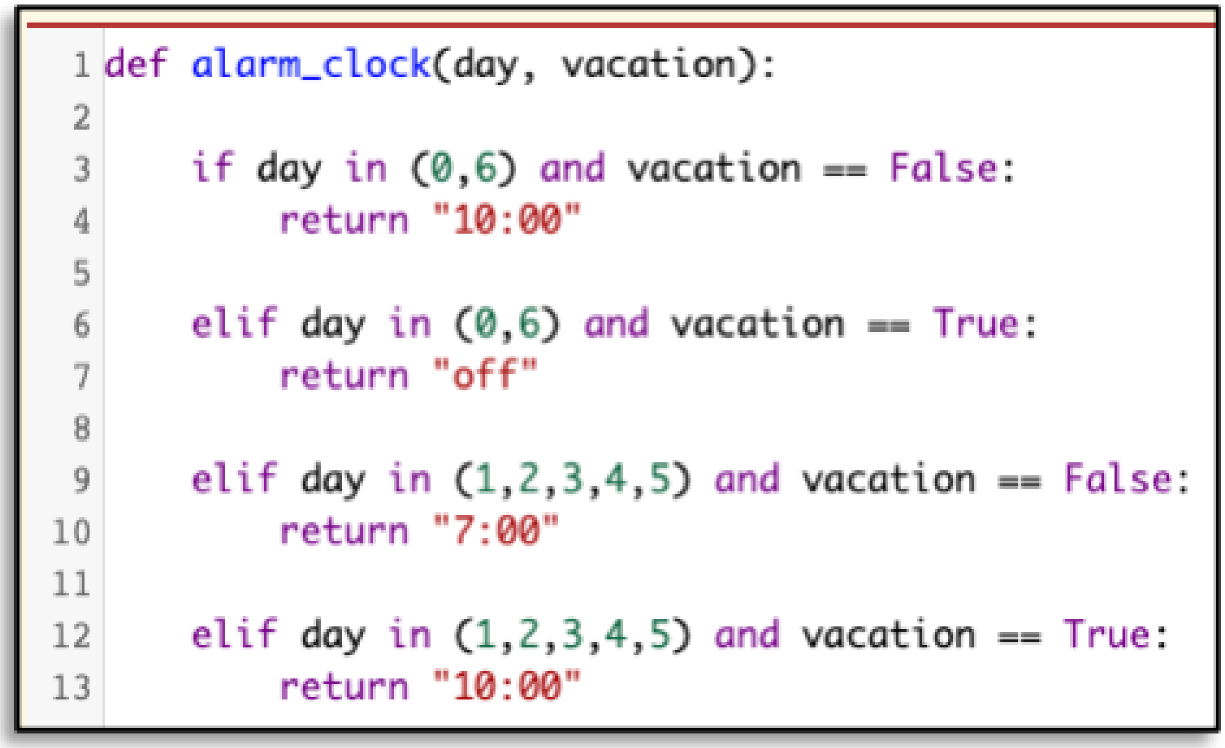


Q-1: Put the code blocks below in order to solve the following problem. There are two extra blocks that are not needed in a correct solution. Given a day of the week encoded as 0=Sun, 1=Mon, 2=Tue, ...6=Sat, and a boolean indicating if we are on vacation, return a string of the form "7:00" indicating when the alarm clock should ring. Weekdays, the alarm should be "7:00" and on the weekend it should be "10:00". Unless we are on vacation -- then on weekdays it should be "10:00" and weekends it should be "off".

```
1   else:
2a  if (day == 0) || (day == 6):
2b  if day == 0 or day == 6:
3   return 'off'
4   return '10:00'
5   if day == 0 or day == 6:
6   else:
7   return '7:00'
8   return '10:00'
9   else:
10a def alarm_clock(day, vacation):
10b def alarm_clock(day, vacation)
11  if vacation:
```

```
2b  def alarm_clock(day, vacation):
4       if vacation:
6           if day == 0 or day == 6:
10              return 'off'
9           else:
3               return '10:00'
5       else:
1b          if day == 0 or day == 6:
11              return '10:00'
7           else:
8               return '7:00'
```

Figure 3.5: *Parsons Problem with a Higher Median Time to Solve than the Equivalent Write-Code Problem (Problem 3 in Table 2).*



```python
def alarm_clock(day, vacation):

    if day in (0,6) and vacation == False:
        return "10:00"

    elif day in (0,6) and vacation == True:
        return "off"

    elif day in (1,2,3,4,5) and vacation == False:
        return "7:00"

    elif day in (1,2,3,4,5) and vacation == True:
        return "10:00"
```

Figure 3.6: *A Student Solution to Problem 3 that used Four Complex Conditionals.*

CHAPTER 4

Problem-Solving Efficiency and Cognitive Load of Parsons vs. Write-Code Problems

Novice programmers need differentiated assessments (such as adaptive Parsons problems) to maximize their ability to learn how to program. Parsons problems require learners to place mixed-up code blocks in the correct order to solve a problem. A within-subjects experiment was conducted to compare the efficiency and cognitive load of solving adaptive Parsons problems versus writing the equivalent (isomorphic) code. Undergraduates were more significantly efficient at solving a Parsons problem than writing the equivalent code, but not when the solution to the Parsons problem was unusual (i.e., it did not match the most common student written solution). This has implications for problem creators. This chapter also reports on the mean cognitive load ratings of the two problem types and the relationship between efficiency and cognitive load ratings. Lastly, it reports on think-aloud observations of 11 students solving both adaptive Parsons problems and write-code problems and the results from an end-of-course student survey.[1]

4.1 Introduction

Novice programmers require well-curated instruction and assessment informed by equity, diversity, and access initiatives to conquer historical issues associated with completing introductory computer programming courses successfully. Some of these issues include high dropout and failure rates [25, 402, 27], the acquisition and retention of basic programming knowledge (i.e., the fragility of programmers' knowledge), and the recruitment and reflection of society writ large concerning social markers (i.e., race, gender, dis/ability, sexuality, etc.) [311].

Learning to program is difficult [90, 312]. Novice programmers experience anxiety and frustration because of their unfamiliarity with syntactic and semantic errors [208, 228, 311]. Deliberate practice improves knowledge acquisition and retention (expertise) [105]. Many programming

[1]Portions of this chapter were adapted from [153]

courses require students to practice mostly by writing code from scratch, but this can take an unexpectedly large amount of time [23] and students can get stuck [187]. Repeated failures, especially when first learning, lower self-efficacy for programming [188]. Women and students from groups that are underrepresented in computing typically have less prior programming experience, which makes them more likely to be negatively affected by practice that relies on writing programs from scratch [183, 220, 175]. To improve diversity, learning tasks should be designed to challenge but not frustrate students and should help struggling students succeed [215]. Computer science programming practice imbued with an inclusive pedagogical approach, such as adaptive Parsons problems, could support learners with a broad range of academic abilities and prior experience [170].

Several researchers have explored the use of Parsons problems to develop basic programming knowledge [89, 102, 283]. Parsons problems are a type of code completion problem in which learners are required to place lines of mixed-up code blocks in the correct order [283]. Some Parsons problems also require the learner to properly indent the code blocks [165]. Parsons problems can also have distractors, which are extra code blocks that are not needed in the correct solution [283]. Adaptive Parsons problems modify the difficulty of the current or next problem based on the learner's performance [203, 100, 98]. Parsons problems may lower cognitive load in comparison to writing code from scratch [312]. They can be used as formative or summative assessments [79, 102], although they may not be perceived as being as authentic as creating a program without scaffolding [338]. Providing novice programmers with differentiated instruction and assessment (i.e., both adaptive Parsons problems and equivalent write-code problems) could increase the likelihood of success for students across academic abilities [365].

Studies have provided evidence that solving (both non-adaptive and adaptive) Parsons problems with distractors is more efficient and equally effective as writing the equivalent code for learning gains from pretest to posttest [102, 100]. However, these studies were conducted in controlled conditions, not as part of a regular course, and used a between-subjects design. Researchers recommend that further studies investigate the effectiveness of Parsons problems [89] and suggest testing the hypothesis that self-reported cognitive load is less for solving Parsons problems than for the equivalent write-code problems [100, 98]. Since cognitive load is dependent on the learner, we chose a within-subjects design. Furthermore, Helminen et al. posit that behavioral analyses of problem-solving strategies are more meaningful than simply examining the outcome and suggest researchers use think-aloud protocols to understand what programmers are thinking [156]. Students' attitudes about assessments are linked to both negative and positive educational outcomes [44, 42]. Thus, we also examined student preferences for different features. The research questions were:

RQ1: What is the effect on efficiency of solving adaptive Parsons problems with distractors versus writing the equivalent code?

RQ2: What is the effect on cognitive load of solving adaptive Parsons problems with distractors versus writing the equivalent code?

RQ3: How does efficiency relate to cognitive load?

RQ4: What are undergraduate students' attitudes towards adaptive Parsons problems versus write code assessments?

We found that (1) the median time to solve each adaptive Parsons problem was less than that of the equivalent write-code problem and the difference was significant for four out of five problems, (2) the problem that was not significantly faster to solve as a Parsons problem than as a write-code problem used a solution that was unusual, (3) there was evidence of an ordering effect in that students who solved the Parsons problem first, with an unusual solution, were much more likely to use that solution in the write-code problem than those who wrote the code first, (4) students in the think-aloud study all perceived that solving an adaptive Parsons problem was easier than writing the equivalent code and evidence from the self-reported cognitive load questions supports that, but students thought that they learned more from writing code as long as they did not get stuck, (5) students found the adaptation helpful when it removed distractors or combined blocks that were not already together, but were confused when it provided indentation or combined blocks that were already adjacent, and (6) while most students found adaptive Parsons problems helpful for learning, some students had a strong negative reaction to Parsons problems and would rather write the equivalent code.

These findings provide evidence that solving an adaptive Parsons problem is significantly more efficient than writing the equivalent code usually, but not always. This implies that to maximize efficiency, the solution to a Parsons problem should match a common student solution. The ordering effect provided evidence that Parsons problems can be used to teach students new ways of solving problems. Students understood most of the adaptation process, but not all of it, which suggests ways to improve the process. And, finally, while most students find Parsons problems valuable for learning, some do not, which implies that students should be given the choice to solve either a Parsons problem or the equivalent write-code problem with unit tests.

4.2 Method

4.2.1 Research Design

We used both a within-subjects experiment and a think-aloud study with questions at the end of each session in this research.

4.2.2 Context

The studies were conducted at a large public research university in the northern United States. All participants were enrolled in a data-oriented programming course in Python during the winter semester of 2020 (N = 152).

This course is the second Python course for School of Information majors, though other majors take it as well. It requires prior programming experience. The course focuses on developing intermediate programming skills in Python and covers working with data from a variety of sources (strings, files, APIs, websites, and databases), object-oriented programming basics, regular expressions, debugging, testing, and SQL.

Dr. Barbara J Ericson created a set of five practice problems for most lectures to provide more active learning during lecture. Active learning has better learning gains and is more motivating than passive learning (traditional lecture) [38, 61, 118]. These problems were typical of those in introductory programming courses. Some of the problems came from past Advanced Placement (AP) Computer Science (CS) A exams which secondary students take for college credit and/or placement and some from the CodingBat website created by Nick Parlante of Stanford University [282]. The AP CS A course is intended to be equivalent to a first course for computer science majors at the college level. Most of these problems were adaptive Parsons problems since it would be hard to individually help students who struggle to solve write-code problems in a large lecture hall. Students earned a point for solving each problem correctly.

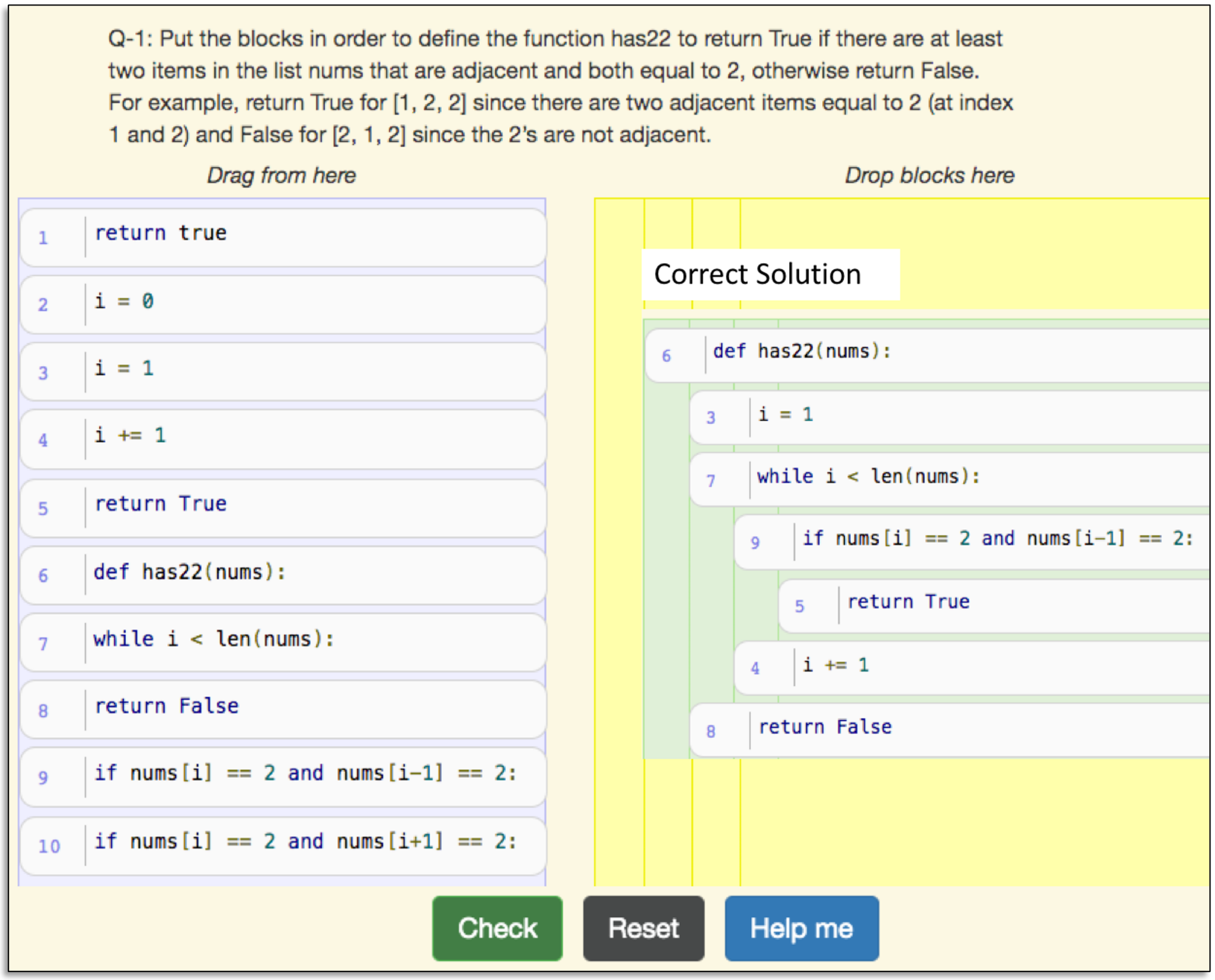
For the course demographics (N = 152), 49% identified as female and 51% as male, the ages ranged from 18 to 36 years old. Eleven percent identified as Black or Hispanic, 36% as Asian, 42% White, 8% as Multiracial, and 4% did not indicate their race. The average GPA was 3.433 (SD = 0.419).

We had IRB permission to analyze anonymous log files from our interactive ebooks. We also received student consent from 95 (61%) students to use their grade and demographic data for research. As part of that consent form, students could also agree to be contacted for an observation. Forty-five percent of the participants who gave consent (n = 95) identified as female and 55% as male; the ages ranged from 18 to 24 years old (M = 20 years, SD = 1.09). Eight percent identified as Black or Hispanic, 35% Asian, 45% White, 8% multiracial, and 4% did not indicate their race.

Sixteen percent were Computer Science (CS) majors and 84% were non-CS majors. The average GPA was 3.871 (*SD* = 0.407).

4.2.3 Materials

We developed an A and B version of a problem set where the only difference between the versions was the problem type. In version A, the first problem was a Parsons problem, and in version B, the same problem was presented as a write-code problem as shown in Figures 4.1 and 4.2.



Q-1: Put the blocks in order to define the function has22 to return True if there are at least two items in the list nums that are adjacent and both equal to 2, otherwise return False. For example, return True for [1, 2, 2] since there are two adjacent items equal to 2 (at index 1 and 2) and False for [2, 1, 2] since the 2's are not adjacent.

Drag from here

```
1   return true
2   i = 0
3   i = 1
4   i += 1
5   return True
6   def has22(nums):
7   while i < len(nums):
8   return False
9   if nums[i] == 2 and nums[i-1] == 2:
10  if nums[i] == 2 and nums[i+1] == 2:
```

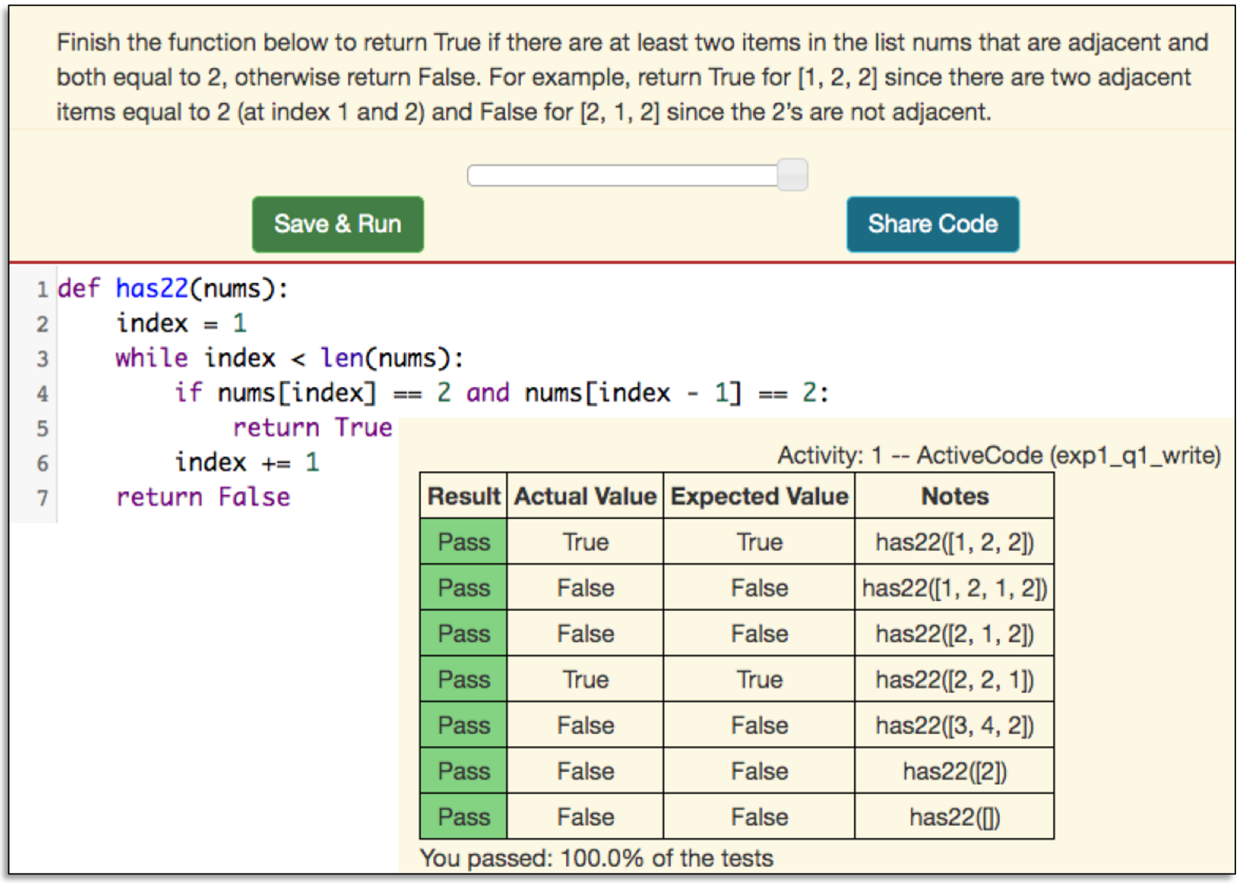
Drop blocks here

Correct Solution

```
6   def has22(nums):
3       i = 1
7       while i < len(nums):
9           if nums[i] == 2 and nums[i-1] == 2:
5               return True
4           i += 1
8       return False
```

Figure 4.1: *First problem in Version A as a Parsons problem (Problem 1 in Table 2).*

Both versions contained five problems in the order shown in Table 4.2. The problems were created to be of varying difficulty from easy to hard to vary the expected cognitive load. They covered strings, lists, ranges, conditionals, loops, dictionaries, and functions. The type of problems in each set is shown in Table 4.1. After each problem, participants were asked to complete the Paas


Finish the function below to return True if there are at least two items in the list nums that are adjacent and both equal to 2, otherwise return False. For example, return True for [1, 2, 2] since there are two adjacent items equal to 2 (at index 1 and 2) and False for [2, 1, 2] since the 2's are not adjacent.

```python
def has22(nums):
    index = 1
    while index < len(nums):
        if nums[index] == 2 and nums[index - 1] == 2:
            return True
        index += 1
    return False
```

Activity: 1 -- ActiveCode (exp1_q1_write)

Result	Actual Value	Expected Value	Notes
Pass	True	True	has22([1, 2, 2])
Pass	False	False	has22([1, 2, 1, 2])
Pass	False	False	has22([2, 1, 2])
Pass	True	True	has22([2, 2, 1])
Pass	False	False	has22([3, 4, 2])
Pass	False	False	has22([2])
Pass	False	False	has22([])

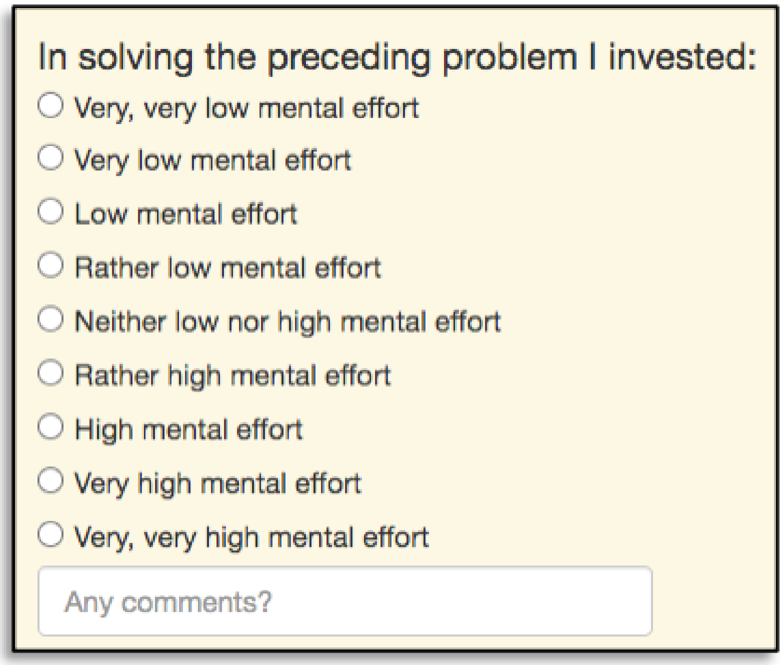
You passed: 100.0% of the tests

Figure 4.2: *First problem in Version B as a write-code problem (Problem 1 in Table 2).*

cognitive load scale [271]. This questionnaire uses a 9-point Likert scale that asks participants to rate the investment of mental effort in solving the previous problem from *"very, very low mental effort"* to *"very, very high mental effort"* as shown in Figure 4.3 (see Appendix C).

Table 4.1: Order and Type of Problem by Version

Version	Problem Type
A	Parsons, Write, Parsons, Write, Parsons
B	Write, Parsons, Write, Parsons, Write


In solving the preceding problem I invested:

- ○ Very, very low mental effort
- ○ Very low mental effort
- ○ Low mental effort
- ○ Rather low mental effort
- ○ Neither low nor high mental effort
- ○ Rather high mental effort
- ○ High mental effort
- ○ Very high mental effort
- ○ Very, very high mental effort

Any comments?

Figure 4.3: *The Cognitive Load Question.*

4.3 Within-Subjects Experiment

4.3.1 Study Design

We conducted a within-subjects field experiment to test the efficiency (time to first correct solution) and self-reported cognitive load of solving adaptive Parsons problems with distractors that contain semantic and syntactic errors versus writing the equivalent code.

At the end of the course (week 15), participants were asked to complete both versions (A and B) of the problem set as an extra credit assignment. If the day they were born was even, they were asked to start with version A and then complete version B, and if they were born on an odd day they were asked to start with version B and then version A. They earned one point for completing each problem and cognitive load question successfully for a total of 20 possible points or 1% of the grade. Students had to earn 2,000 points or more to receive an A+ in the class.

4.3.2 Participants

This section is reports on data from the students who completed a problem in the problem set as both a Parsons problem and a write-code problem. We are only reporting the data from students who permitted us to use their data to for research ($n = 95$). Since this was extra credit and students received a point for correctly completing each problem, the number of students who correctly completed both types of a problem ranged from 37 to 46 as shown in Table 4.2.

4.3.3 Analysis

We downloaded a log file for the course and only kept the data from the students who had given consent to use their data for research.

We calculated the time in seconds to the first correct solution for both the Parsons problems and write-code problems for each student, using the anonymized id. We compared the medians to account for those who had no interaction for more than five minutes or who spent time on other problems before returning to previous ones. In our think-aloud sessions, we found that one student wanted to take a break from solving a hard problem. She switched to another problem and then later came back to solving the hard problem.

We ran Mann-Whitney U tests with continuity corrections to analyze the difference in the median times to the first correct solution within the groups since this was a within-subjects study and the data violated assumptions of normality and equal variances [260]. We report probability-based effect sizes for this analysis as this measure is reported to be more robust when parametric assumptions are violated [66, 323]. We computed these using the R package canprot. We ran paired *t*-test to analyze the difference between cognitive load ratings within groups because this data met the assumptions of normality and equal variances [164]. We report Cohen's d_{rm} effect sizes for this analysis, computed using the R package lsr [264], as this measure is recommended over others for repeated measures data [209]. Multiple comparisons were adjusted for using Bonferroni's correction [59]. Finally, we ran Spearman correlation coefficient tests to analyze the relationship between the times to the first correct solution and cognitive load ratings within groups [149].

4.3.4 Results

4.3.4.1 Efficiency

The results are shown in Table 4.2. Notice that the median time to solve each Parsons problem was always less than the median time to write the equivalent code. For four of the five problems, the difference was statistically significant. It was not significant on problem one shown in Figure 4.1, which was intended to be one of the more difficult problems. Interestingly, it was significant for one of the problems that were intended to be easier (problem three). This function returns the difference between the maximum and minimum values in a list. It was most significant for problem five as a Parsons problem (see Figure 4.4), which was intended to be one of the harder problems.

The probability-based measure of effect sizes (*A*) shown in Table 4.2 means that the probability ranges from 69% to 92% that a randomly selected student will solve one of the problems that had

Table 4.2: Time to Complete Parsons Problem vs. Write-Code Problem

		Parsons Problem	Write-Code Problem	Mann-Whitney U		
Problem (Diff.)	*n*	*Mdn* in seconds	*Mdn* in seconds	*V*	*p*-value	*A*
1 has22 (H)	40	157.5	166.5	274	$p = 0.343$	0.58
2 countInRange (M)	43	102	187	190	$p = 0.003$**	0.69
3 diffMaxMin (E)	42	14	34	145.5	$p < 0.001$***	0.78
4 dictTotal (M)	46	25.5	44.5	205.5	$p = 0.002$**	0.73
5 dictNames (H)	37	66	216	5	$p = 0.002$**	0.92

Note: E = Easy, M = Medium, H = Hard; * p $<$ .05, ** p $<$ .01, *** p $<$ .001; *A* = probability-based effect size measure (nonparametric generalization of common language effect size statistic).

a significant difference between completion times faster as an adaptive Parsons problem than as a write-code problem.

4.3.4.2 Cognitive Load Ratings

The mean self-reported rating of mental effort (cognitive load) was lower for each problem as a Parsons problem than as a write-code problem as shown in Table 4.3. However, the difference was only statistically significant for two of the problems (four and five). There was a large effect size on problem five (0.94) shown in Figure 4.4, which was also the problem that had the largest median time to solve as a write-code problem. This problem was intended to be one of the more difficult problems.

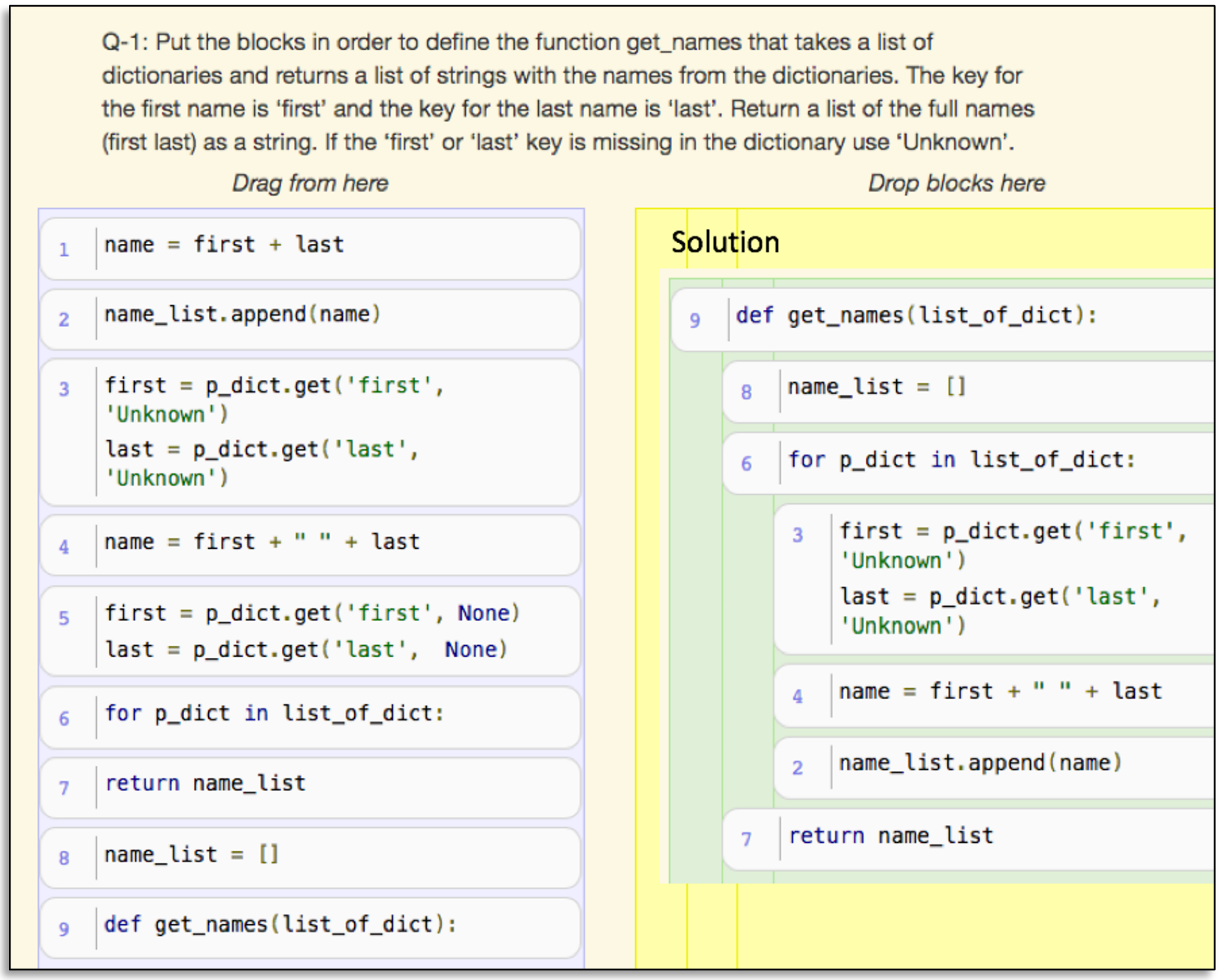
Table 4.3: Cognitive Load Ratings for Parsons Problems vs. Write-Code Problems

	Parsons Problem	Write-code Problem	Paired *t* test				
Problem (Diff.)	*M (SD)* of ratings	*M (SD)* of ratings	*t* value	*df*	*p*-value	Cohen's d_{rm}	*A*
1 has22 (H)	3.20 (1.62)	3.65 (1.79)	-1.4327	39	$p = 0.800$	0.23	0.51
2 countInRange (M)	3.02 (1.34)	3.49 (1.84)	-1.5496	42	$p = 0.644$	0.24	0.50
3 diffMaxMin (E)	1.05 (1.31)	1.45 (1.42)	-2.0058	41	$p = 0.258$	0.31	0.45
4 dictTotal (M)	1.52 (1.56)	2.28 (1.86)	-2.8044	45	$p = 0.037$*	0.41	0.53
5 dictNames (H)	2.05 (1.43)	3.70 (1.70)	-5.7257	36	$p < 0.001$***	0.94	0.71

Note: E = Easy, M = Medium, H = Hard; * p $<$ .05, ** p $<$ .01, *** p $<$ .001, paired t-test; Likert scale: 1 = Very, very low mental effort; 2 = Very low mental effort; 3 = Low mental effort; 4 = Rather low mental effort, 5 = Neither low nor high mental effort; 6 = Rather high mental effort; 7 = High mental effort; 8 = Very high mental effort, 9 = Very very high mental effort

4.3.4.3 Efficiency and Cognitive Load Ratings

Results showed there were weak positive and negative relationships between Parsons problems efficiency and cognitive load ratings. The write-code problems, one, two, and four, had significantly


Q-1: Put the blocks in order to define the function get_names that takes a list of dictionaries and returns a list of strings with the names from the dictionaries. The key for the first name is 'first' and the key for the last name is 'last'. Return a list of the full names (first last) as a string. If the 'first' or 'last' key is missing in the dictionary use 'Unknown'.

```
1  name = first + last

2  name_list.append(name)

3  first = p_dict.get('first',
   'Unknown')
   last = p_dict.get('last',
   'Unknown')

4  name = first + " " + last

5  first = p_dict.get('first', None)
   last = p_dict.get('last',  None)

6  for p_dict in list_of_dict:

7  return name_list

8  name_list = []

9  def get_names(list_of_dict):
```

```
9  def get_names(list_of_dict):
8      name_list = []
6      for p_dict in list_of_dict:
3          first = p_dict.get('first',
           'Unknown')
           last = p_dict.get('last',
           'Unknown')
4          name = first + " " + last
2          name_list.append(name)
7      return name_list
```

Figure 4.4: *Parsons Problem Five in Table 2.*

moderate positive relationships with efficiency and cognitive load ratings. For every other write-code problem, there was a weak positive relationship between efficiency and cognitive load ratings (see Table 4.4).

4.4 Think-Aloud Study

We conducted a think-aloud study to get an in-depth look at what students were thinking as they solved adaptive Parsons problems and write-code problems and to see where they struggled and why. We used the same problems that were used in the within-subjects study. At the end of each session, we asked three open-ended questions to probe student's perceptions, understanding, and desire for help while writing code.

Table 4.4: Correlations between Completion Time and Cognitive Load Ratings

	Parsons Problem			Write-code Problem		
Problem	*n*	*ρ*	*p*-value	*n*	*ρ*	*p*-value
1 has22	61	.12	0.34	41	.34	0.03*
2 countInRange	53	.07	0.61	52	.47	0.00***
3 diffMaxMin	62	.13	0.30	45	.10	0.50
4 dictTotal	54	-.04	0.77	56	.28	0.04*
5 dictNames	58	.03	0.80	42	.28	0.07

Note. * $p < .05$, ** $p < .01$, *** $p < .001$, ρ = Spearman rank-order correlation coefficient

4.4.1 Participants

Sixty-one students from the course consented to be contacted for a think-aloud study. They were contacted via email in week 12. Eleven students participated. The participants were between 19 and 21 years old. There was one Black student, one multiracial student, three Asian students, and four White students. Their average GPA was 3.283 out of a 4.0.

Participants were randomly assigned to either version A or B. Our protocol required them to verbalize their thoughts as they worked through each problem [393]. Five (45%) participants completed version A and six (55%) completed version B. Each session was conducted on-line via BlueJeans or Zoom and recorded with verbal consent from the participant. The sessions lasted on average thirty-one minutes and eleven seconds. One session lasted for one hour, eleven minutes, and twenty-one seconds.

The three questions that we asked the participants to answer at the end of each session were: (1) *Which do you prefer: writing code or solving mixed up code problems and why?* (2) *Have you used the "Help Me" button on the mixed-up code problems? Did you find it helpful? Was there anything that happened that you didn't understand?* and (3) *Would you like to see a "Help Me" button for write-code problems?* The "Help Me" button triggers intra-problem adaptation. It either removed a distractor from the solution, provided the indentation, or combined two blocks into one.

4.4.2 Analysis

The think-aloud sessions were recorded and transcribed. Two researchers reviewed the videos and transcripts and provided a summary of each session. The summary was used to identify sessions to review in more detail. The researchers also tallied the responses to the open-ended questions and identified candidate quotes for the paper.

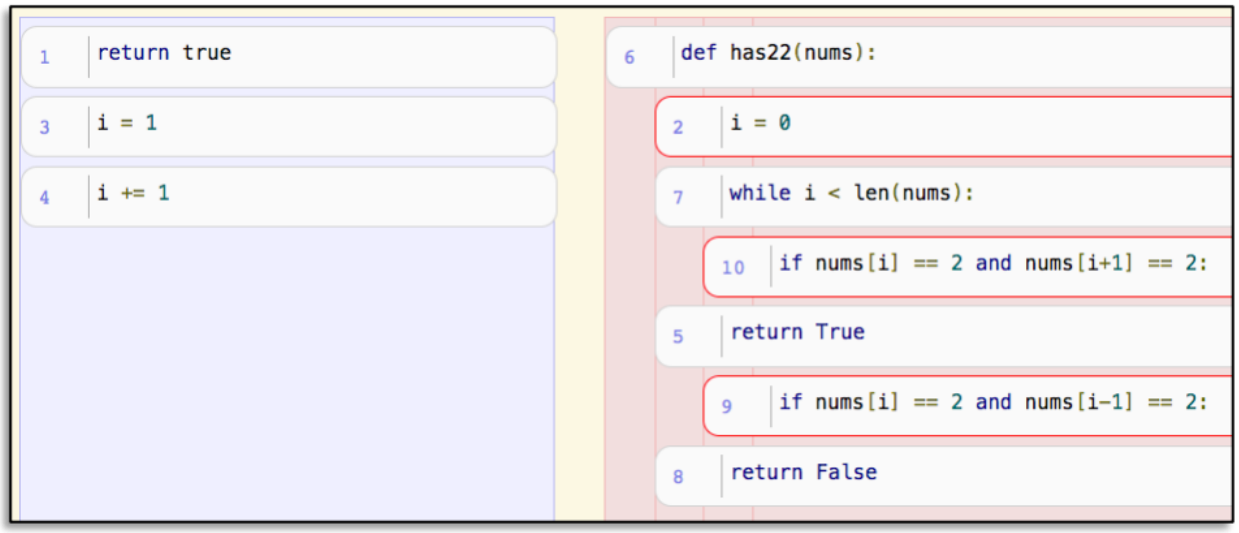
4.4.3 A Deeper Dive into Problem One

This section explores the difficulties students had with problem one as shown in Figure 4.1, since that was the only problem without a statistically significant difference in the time to solve it as a Parsons problem versus as a write-code problem. This problem involved a function **`has22`** that took a list of numbers, **`nums`**, and returned **`True`** if it found at least two adjacent 2's in the list and **`False`** otherwise as shown in Figure 4.1.

The given solution to the Parsons problem can be seen in Figure 4.1. There were three distractors. The first distractor (block 2) in Figure 4.1 initialized the variable **`i`** to zero which would have been correct if the student also used the second distractor (block 10), however the only loop given was a while loop (block 7) that looped while **`i`** was less than the number of items in the list **`nums`**. Using block 10 would have resulted in an out of bounds error since it would try to check the last number in the list against a number past the end of the list. The other distractor was block 1 which simply used the wrong case for the Python keyword **`True`**. None of the students in the think-aloud study used this distractor. This is not surprising, since this was near the end of their course and they had used these Python keywords many times over the semester.

Two of the distractors, blocks 2 and 10, distracted several of the students. Female subject 1 said, *"So, I'm going to initialize **`i`** as zero (using distractor 2), so then when I use my while loop, because I'm assuming I'm going to index and indexing starts at zero."* She is correct that the first index is zero in a list, but this solution requires **`i`** to be initialized to one. Later she realized her mistake, *"Actually, I need to start this **`i`** at one. Since my if statement indexes at **`i`** minus one. And I need that **`i`** minus one, to actually be zero."* Male subject 3 also had trouble with these distractors, but he also had several other problems. He had to stop and look up while loops online because he was less familiar with them. Most of the problems in the course used for loops rather than while loops. He took some time to read about while loops on W3Schools. His first solution used two distractors (blocks 2 and 10) as shown in Figure 4.5. Interestingly, when he talked through what he was thinking he claimed that block 10 tested if there were 2s that were adjacent, but also said he assumed that block 9 tested if the 2s were not adjacent, even though block 10 is actually another way to test if there are adjacent 2s. His solution had several errors. The return statements (blocks 5 and 8) were not correctly indented under the conditionals, even though as he talked through the code it was clear that he intended them to be. The solution does not include block 4 which increments **`i`**, so it would not loop correctly. Another problem in his logic was an immediate return if adjacent 2s were not found, rather than looping through the rest of the list to check the rest of it.

He then replaced block 2 with block 4 and switched blocks 9 and 10 as shown in Figure 4.6. He did not realize that you can not increment **`i`** if you have not initialized it. He also said that he just assumed that he got blocks 9 and 10 mixed up since they were highlighted as being wrong or

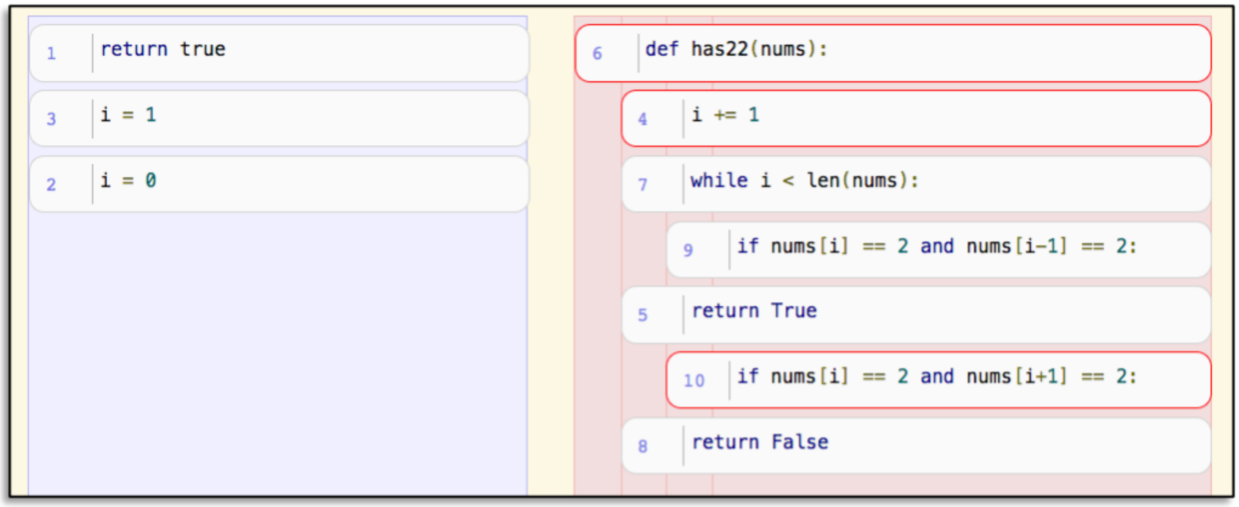


```
1  return true
3  i = 1
4  i += 1
```

```
6  def has22(nums):
2      i = 0
7      while i < len(nums):
10         if nums[i] == 2 and nums[i+1] == 2:
5      return True
9          if nums[i] == 2 and nums[i-1] == 2:
8      return False
```

Figure 4.5: *First Attempt at a Solution for Problem 1.*

in the wrong order. He was still assuming that one of those blocks checked for adjacent twos and the other tested that they were not adjacent.



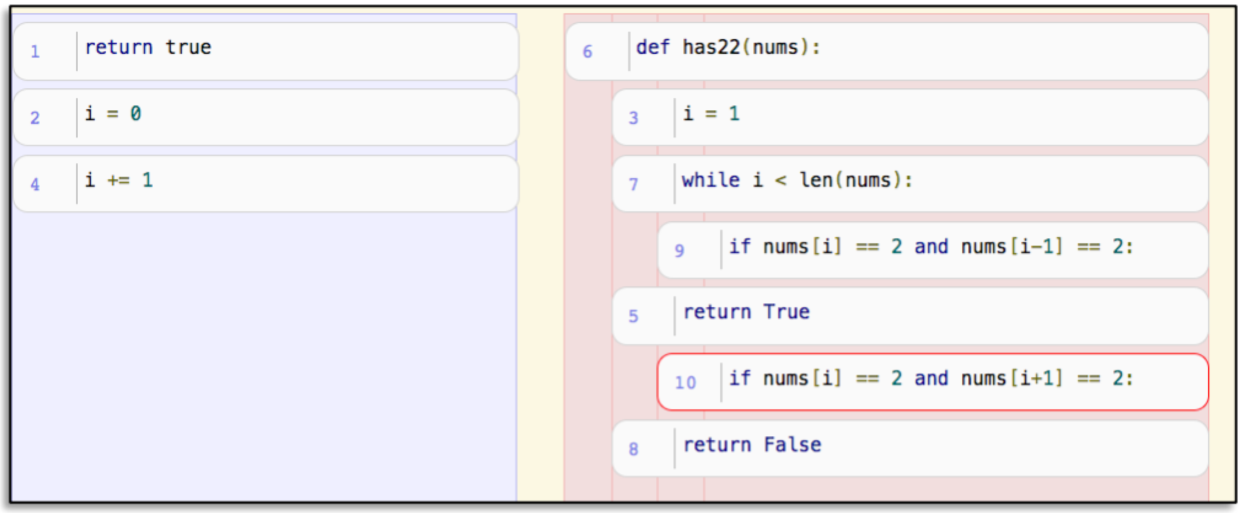
```
1  return true
3  i = 1
2  i = 0
```

```
6  def has22(nums):
4      i += 1
7      while i < len(nums):
9          if nums[i] == 2 and nums[i-1] == 2:
5      return True
10         if nums[i] == 2 and nums[i+1] == 2:
8      return False
```

Figure 4.6: *Second Attempt at a Solution for Problem 1.*

He then replaced block 4 with block 3 and checked that solution which highlighted block 10 as being wrong or in the wrong order. He was also told that help was available if he wanted to make the problem easier.

He was not sure why block 10 was wrong but clicked the "Help Me" button which triggered the intra-problem adaptation and block 10 (a distractor) moved back to the source on the left and grayed out. He said, *"Oh, okay."* He checked the current solution and received feedback that



```
1  return true
2  i = 0
4  i += 1
```

```
6  def has22(nums):
3      i = 1
7      while i < len(nums):
9          if nums[i] == 2 and nums[i-1] == 2:
5      return True
10         if nums[i] == 2 and nums[i+1] == 2:
8      return False
```

Figure 4.7: *Third Attempt at a Solution for Problem 1*

his solution was too short. It was still missing block 4. He said he didn't understand why. He clicked the "Help Me" button again and it provided the indentation, which fixed his problems with indentation. He checked that solution and it still said the solution was too short. He said, *"I don't quite understand."* He then said, *"So now I am just going to guess in all honesty."* He moved block 4 to the correct location and checked the solution and it was correct. He said, *"So that worked"* and laughed.

He rated the problem as a 5 on the cognitive load scale which means neither low nor high mental effort. He never tried to simulate the code execution on any of the examples to understand what was happening and what could be wrong. His focus was on getting the problem correct, but not on understanding why the solution was correct. He might have benefited from an explanation of why the distractor was incorrect or from the ability to step through the correct code.

4.4.4 Interview Question Results

In this section, we report on students' answers to each of the three open-ended interview questions.

4.4.4.1 Preference

Overall, participants found Parsons problems to be easier than the write-code problems. However, they felt that they learned more from write-code problems unless they got stuck while trying to solve them. When asked which type of problem do you prefer and why, eight (72.7%) participants said they preferred write-code problems. They reported that write-code problems help them understand syntax and semantic errors and that solving them leads to a greater sense of accomplishment.

They also appreciated that they could solve the write-code problems in a variety of ways. One participant said *"The beauty of the write-code [problems are] that you have to create your own solutions"* (Female Subject 1). One of the drawbacks of Parsons problems is that they can only have one correct solution and that solution may not match the student's way of solving the problem. One student said, *"I guess the only like difficulty or confusion with [Parsons problems are their] naming conventions. I guess I wouldn't name everything the same way that they name it"* (Female Subject 4).

Two (18.2%) participants strongly disliked Parsons problems. One participant said:

> *"Personally, I hate these block code problems. I think, more so than anything, it's like a waste of time. I don't think very many people, at least the people that I know and the people that I do work with in class, I don't think a lot of people like them. I think people like them because they're easy, not because they're actually beneficial in any way"* (Male Subject 9).

Both of the participants who strongly disliked Parsons problems had prior programming experience. Male Subject 10 said:

> *"...I came into this class with a little bit more Python experience because I just had an internship....I still think that it's a better exercise to make people write-code in long form... because that's what you're going to end up doing when you're actually solving problems. You'll have bits and pieces that may or may not be wrong. I think [solving Parsons problems is] not constructive to learning. And also, to be honest [about] the system, you can just kind of drag and guess and then just like click "Help Me". And that's what most people do."*

The "Help Me" button triggers intra-problem adaptation. Each time the "Help Me" button is clicked it will first remove a distractor if there are any in the solution. If there are no distractors in the solution, it next provides indentation. If indention has been provided, it next combines two blocks into one. It combines blocks until three blocks are left in the solution. If the "Help Me" button is clicked when there are only three blocks in the solution, it tells the student that they should be able to solve the problem. While this student suggests that many students used the "Help Me" repeatedly till the problem was correct, a log file analysis showed that an average of only 7.3% of students who used the "Help Me" button did this.

In contrast, some participants found Parsons problems helpful for learning how to program, especially when they lacked prior knowledge of the basics. They liked that Parsons problems provided them with the code so that they could focus on other tasks. One participant said *"[I prefer] the [Parsons] problems because they allow me to focus on how certain problems should* *be structured and offer more help in the process. If I get stuck [on write-code] problems, it is very hard to see where I am going wrong"* (Female Subject 6). Another student said the following when solving the first problem as a Parsons problem, *"So I really enjoyed these practice problems (Parsons problems) ... because there's a lot of room for error. If you really don't know, you're able to really just plug in and see what works and everything. But that's kind of just like the lazy way of doing it. The other way would be to certainly just digest it and sometimes it can be kind of hard cause I feel like there's so many options that you have to really pay attention and understand like what's being asked of you when you're trying to solve the problem"* (Male Subject 3). One student, who got stuck on the first write-code problem, but solved the next Parsons problem said that she would rate the write-code problem an eight and the Parsons problem a one or two: *"It's not the fact that [the write-code problem is) necessarily hard. But it's the fact that I'm missing, something and since I don't know what I'm missing it's going to be hard for me to figure it out"* (Female Subject 8).

While solving the problem set, participants also reported that they prefer to take breaks from solving certain problems and return to them later—a way to self-manage cognitive load. Female Subject 8 said, *"...sometimes you just need to take a break from the code and look at it with fresh eyes and you can see exactly what you did wrong."* After noting this, we used the median to compare the time to first correct solution to minimize the effect of this behavior.

4.4.4.2 Parsons Problems and Help-Seeking Features

Participants were also asked if they had ever used the "Help Me" button (intra-problem adaptation) to solve Parsons problems and, if so, did they find it helpful, or was there anything they did not understand. During the think-aloud sessions, few participants used the "Help Me" button. Most participants who had used it during the course found it helpful when it led to distractors being removed or blocks being combined, but they reported being confused when it provided indentation. One student said, *"[The] indentation was kind of confusing because it gets rid of the actual indentation. I don't know, I just think this was kind of weird"* (Female Subject 7). Furthermore, they reported that the help was not useful when it provided something that they already understood or had correct. One participant said *"I use the help button a lot and find it helpful. The only flaw is when [it helps] with a part of the code that I already have correct"* (Female Subject 6). The intra-problem adaptation process first removed any distractors in the solution, then provided indentation (even if the indentation was already correct), and then would pick two blocks to combine (even if the two blocks were already in the correct order) based on the number of lines in each block.

Due to this feedback, we have modified the intra-problem adaptation process to no longer provide indentation and to pick two blocks to combine that are the furthest apart. We plan to conduct think-aloud observations again in the future and will look for evidence that students find these changes helpful.

One student reported liking that you could use the “Help Me” button to solve the problem and then reset the problem to try and solve it again without help. She said:

> *“...it’s like definitely easier doing it with the blocks. ...I guess when you really don’t know what’s going on they really like make it, I guess like push you in the right direction is like super, super helpful and especially being able to like reset it and do it again yourself without like the “Help Me” function also like it just like ingrains this syntax into my brain more”* (Female Subject 4).

One student reported finding the intra-problem adaptation useful for correctly solving the problem and then working to try to understand that solution, *“So it’s pretty useful for getting the actual problem [correct], I think, and I think it’s kind of my job to understand okay, well, now I have to understand why it’s the right solution ...”* (Male Subject 5). This was one of the original goals for Parsons problems, to be an example of an expert’s solution to a problem. Expert solutions to a problems are also called a worked examples. Research has shown that learning can be improved by studying worked examples, especially if they are followed by similar practice problems [372, 385].

4.4.4.3 Write-code Problems and Help-Seeking Features

Finally, participants were asked if they would like to see a “Help Me” button for write-code problems. Eight (72.7%) participants said they would like to see such a feature because it would both allow them to come up with their own solutions and provide hints as to what to do next. One participant said, *“A “Help Me” button that either highlights the error or provides a hint or feature to apply to the problem would be very helpful for the code writing problems”* (Female Subject 06). Another said:

> *“I think it would be very helpful if there was a help me button. . . .Maybe if it gave you a hint. A piece of code that you’re missing or maybe a hint like use accumulator pattern or use for loop or that you should be using a for and if [statement]. Maybe that. Use one for loop and one if or use the accumulator pattern. Give hints on what you should be using to solve the code or what they think you should be using to solve the code”* (Female Subject 08).

However, some expressed concerns that such a feature would stifle creativity and/or defeat the purpose of these types of questions. One participant said, *“. . . too much help would kind of defeat the purpose of like self-testing or like doing work assignments”* (Female Subject 4). The same participant also reported the “Help Me” button replaced searching for help on sites like W3Schools.com. Two participants used outside resources (W3Schools and Stack Overflow) to help them solve Problem 5.

4.5 Further Analysis of Problem 1

Since problem 1 (has22) was the only problem without a significant difference between the median time to solve it as a Parsons problem versus a write-code problem during the within-subjects study, we investigated it further.

4.5.1 Was the Parsons problem solution unusual?

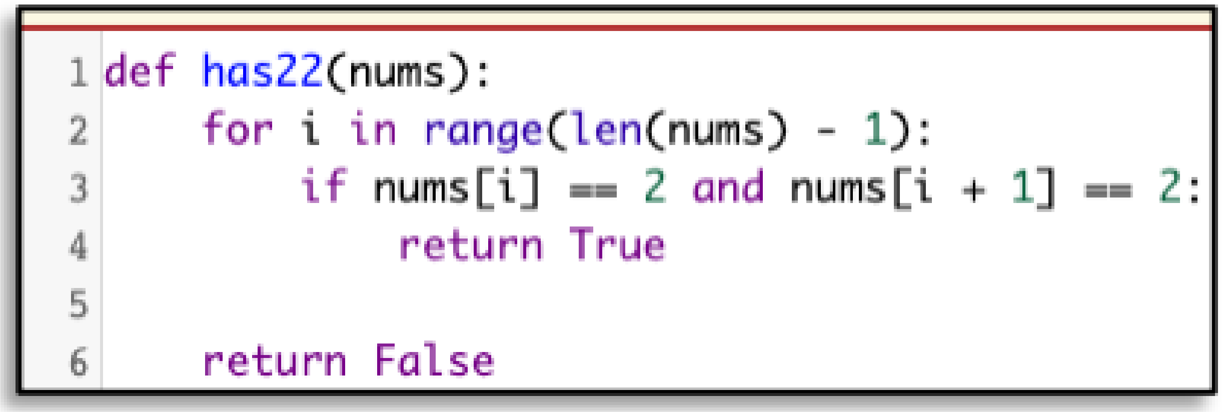
We hypothesized that the provided solution to the Parsons problem was unusual in that it used a while loop and started looping at index one as shown in Figure 4.1 and the think-aloud study supported that hypothesis. The students had more experience with for loops than while loops and with starting a loop at index zero rather than index one. Since we did not need any personal data for this analysis, we used the anonymous data from all of the students from the within-subjects study who wrote code for this problem, rather than just the data from the students who permitted us to use their data for research. Forty-eight of the students (84%) solved this problem first as a Parsons problem while only nine (16%) wrote the code first. Even though we asked students to start with version A if they were born on an even day and B if they were born on an odd day, it seems that many just solved A first and then B. This may have been due to the assignments being visible in the eBook before the announcement was made about how to do the assignment. Of the nine students who wrote the code first, seven (78%) used a for loop and only two (22%) used a while loop.

4.5.2 Was there an ordering effect?

Students who solved the Parsons problem first were much more likely to use the Parsons problem solution when writing the code. Of the 48 students who solved the problem first as a Parsons problem, 39 (81.3%) used a while loop and 8 (16.7%) used a for loop. In addition, 37 (77.1%) used a solution that was equivalent to the Parsons problem solution. This suggests that solving the Parsons problem first had a substantial effect on how students solved the problem when they wrote the code. This implies that Parsons problems can also be used to teach students new approaches to solving problems, however, more experiments should be done to test this hypothesis.

4.5.3 Results from Problem One Midterm from Fall 2020

Since only nine students wrote the code first before solving the Parsons problem, we also included this problem as a write-code problem on the second midterm for the same course during the fall of 2020. We used an anonymous log file for this analysis and found that **none** of the 113 submitted answers used a while loop to solve this problem. In addition, only 13 (11.5%) of the answers started the loop at index one. Most answers that used a for loop started that loop at index zero as seen in Figure 4.8. This provides strong evidence that the given Parsons problem solution was unusual, which could well be why it was not significantly more efficient to solve as a Parson problem than a write-code problem. However, more work should be done to verify this hypothesis.



```python
def has22(nums):
    for i in range(len(nums) - 1):
        if nums[i] == 2 and nums[i + 1] == 2:
            return True

    return False
```

Figure 4.8: *An example common student solution from the second midterm for Fall 2020*

4.5.4 Use of Intra-Problem Adaptation

We also examined how many students attempted each problem, solved it, and used the intra-problem adaptation (triggered by the "Help Me" button) as well as the type of adaptation for each of the Parsons problems as shown in Table 4.5. As you can see from Table 4.5, over half of the students who attempted the first problem used the "Help Me" button (intra-problem adaptation) at least once. This is further evidence that students found this problem difficult, especially when contrasted with problem three, where no students used the "Help Me" button. Also note that even though the problems were of varying difficulty, from easy to hard, the percentage of students who solved it remained high (96% to 100%). This meshes with prior research that showed that users were nearly twice as likely to get adaptive Parsons problems correct than non-adaptive problems. [98]. Indeed, the goal of intra-problem adaptation is to help users solve the Parsons problem when they are struggling.

4.6 Results from Student Survey

The think-aloud study found that some students liked Parsons problems and found them useful while others hated them and would rather write the code themselves. It also provided evidence that

Table 4.5: Use of Intra-Problem Adaptation

			Intra-Problem Adaptation			
Problem (Diff.)	Attempted	Solved (%)	Used Help Me (%)	Rem. Dist.	Indented	Combined blocks
1 has22 (H)	103	99 (96%)	54 (52%)	39	41	24
2 countInRange (M)	83	80 (96%)	18 (22%)	15	14	6
3 diffMaxMin (E)	96	96 (100%)	0 (0%)	0	0	0
4 dictTotal (M)	83	82 (99%)	2 (2%)	2	1	0
5 dictNames (H)	94	90 (96%)	15 (16%)	9	13	9

Note: E = Easy, M = Medium, H = Hard; Rem. Dist = Removed Distractor.

some students did not understand what happened during the intra-problem adaptation process. To determine how common each of these perceptions/issues were, we added a few questions to the end of course student survey which was administered in the last few weeks of the course. Students answered using a five-point Likert scale where one is strongly disagree and five is strongly agree. All students would have received extra points if the response rate reached a specified level (5 points if 85%). However, the response rate was 77.5% (107 out of 138).

Of the respondents, 80.6% agreed or strongly agreed that the "Help Me" button (intra-problem adaptation) was useful for learning, while 5.9% disagreed. In addition, 17.5% of students agreed or strongly agreed that they did not understand what happened when they clicked on the "Help Me" button which triggered the intra-problem adaptation. Less than a third of respondents (30.4%) agreed or strongly agreed that they would rather write-code than solve a mixed-up code problem, while 40% disagreed or strongly disagreed.

Some of the students found Parsons problems valuable. One student wrote in response to an open-ended question about the aspects of the course that they liked best, *"...solving mixed-up code was surprisingly helpful when it came to helping me understand how to write-code."* Some students preferred to write-code their own way rather than solve a mixed-up code problem, *"Mixed-up code was easier once you got the hang of it, but my style is a bit different than the style in those problems, so I would prefer to write-code from scratch instead."*

4.7 Discussion

Prior research under controlled conditions using between-subjects studies provided evidence that solving adaptive Parsons problems is more efficient than writing the equivalent code and just as effective for learning gains from pretest to posttest [100, 98]. In this study we tested the efficiency of solving adaptive Parsons problems versus writing the equivalent code using a within-subjects design where participants solved the same problem as a Parsons problem and as a write-code problem. The median time to solve each Parsons problem was less than the median time to solve each write-code problem. The difference was significant for four of the five problems. It was not significant for problem 1 in Table 4.2 and shown in Figure 4.1. It is interesting to note that this is the Parsons problem with the highest mean cognitive load rating as seen in Table 4.3. The solution to this Parsons problem used a while loop that started at index one. Further investigation provided strong evidence that this was an unusual approach. One of the students in the think-aloud study even had to look up while loops on W3Schools before solving this problem. Most of the students in the think-aloud study used the distractor that initialized `i` to zero rather than one, since they were more familiar with starting a loop at index zero. One of the drawbacks to Parsons problems is that they can only have one correct solution, yet there are many ways to solve a programming problem.

One interesting finding was an ordering effect. Most of the students (77.1%) who solved problem one as a Parsons problem first, used the Parsons problem solution when writing the code. Only 22% of students who wrote the code first used a while loop, while 77% used a for loop. To further determine the most common student written solution, we added this problem to a midterm in the fall of 2020 and none of the 113 students used a while loop to solve this problem. In addition, only 11.5% of the solutions started looping at index one.

We also wanted to determine if the cognitive load for Parsons problems was less than that of writing the equivalent code. We used a within-subjects design to test this since cognitive load is very subjective. The mean self-reported cognitive load rating for each Parsons problem was less than that of the equivalent write-code problem, but the difference was statistically significant for only two of the five problems.

There was no significant correlation between the completion time and self-reported cognitive load rating for adaptive Parsons problems, but there was for three of the five write-code problems. This indicates that we can not use time on task to determine the cognitive load of adaptive Parsons problems.

The questions at the end of the think-aloud observations showed that all students felt that Parsons problems were easier than writing the equivalent code. However, some students did not value solving Parsons problems and two had strong negative reactions to them. These were students with more prior programming experience. This is both consistent with expertise reversal effect from cognitive load theory in which techniques that are used to help novice students can increase the cognitive load for experts [371] and the difficulty learners have with self-reported perceptions of learning [80].

Observations of teachers solving intra-problem adaptive Parsons problems had provided evidence that some teachers found the adaptation that provided the indentation confusing [98]. The undergraduate students in this study also reported finding that adaptation confusing. In addition, they did not value the adaptation when it combined blocks that were already adjacent.

Some students reported overusing the "Help Me" button (intra-problem adaptation) to get to a correct solution. However, a log file analysis found that only an average of 7.3% of the students who used the "Help Me" button did this. One student reported that they would reset the problem after using the "Help Me" button to try the problem again without help. Our goal with adaptive Parsons problems was to help struggling students reach a correct solution, even when they could not get help from others. We hoped that these students might use the reset button to try to solve the problem again without help.

4.8 Limitations and Future Work

This study was conducted in an undergraduate Python programming course at one institution. These results may not generalize to other languages or populations. For example, students with dis/abilities can experience difficulty working in drag-and-drop programming environments [197, 247]. More research should be done to test these results in other languages and with other populations, especially those with special needs.

It is possible that there was an ordering effect—a lack of independence between the completion times for each problem type because of participants' ability to recall the corresponding problem. We mitigated this concern by randomly assigning participants to start with either version A or B. This design supports a comparison between problem type with subjects serving as their own control for a direct comparison [186]. However, few students started with version B, perhaps since the assignment was visible before they received instructions and they naturally started with A.

Efficiency is a multifaceted theoretical construct that can be explained by measuring completion time as well as learning gains [161]. We did not compare the learning gains from solving Parsons problems versus writing the equivalent code in this study. However, prior research has shown equivalent learning gains for undergraduate students from solving Parsons problems versus writing the equivalent code in controlled experiments [102, 100, 425].

Finally, we measured cognitive load using the Paas scale, but objective measures such as eye-tracking should be collected and compared to subjective measures [245, 422]. There is a growing body of research on eye-tracking in computing education research that has led to a better understanding of how people learn to program [267].

4.9 Implications

If our hypothesis is correct that Parsons problems are not significantly more efficient to solve if the provided solution is unusual, it implies that the best way to generate good Parsons problems is to first gather student solutions to write-code problems and use the most common student solution.

We are currently investigating doing this using Abstract Syntax Trees (ASTs) to cluster the student-written code [306]. It would also be useful to gather the most common errors and use those as distractors. Another alternative is to cluster the write-code approaches and use probing questions to determine the approach that a student intends to use and then select a Parsons problem for that student that matches their approach.

The fact that 77% of the students who solved the Parsons problem first used an equivalent solution to the write code problem, even though that solution was unusual, implies that Parsons problems can be used to teach students new ways to solve problems. However, more work should be done to verify that hypothesis.

The mean self-reported cognitive load rating for each Parsons problem was lower than that of each equivalent write-code problem, yet the difference was significant on only two problems. However, students in the think-aloud observation perceived that solving an adaptive Parsons problem is easier than solving the equivalent write-code problem. We plan to test other approaches for measuring cognitive load, such as eye-tracking, and to explore the relationship between cognitive load and other self-theories that may help explain these differences [94]. If we can reliably determine the cognitive load of a problem for a subject, it might be possible to modify the difficulty of the problem dynamically to keep each student in the zone of proximal development. Eye tracking would be less intrusive than asking the students what their mental effort was after each question.

While most students in this course (80.6%) reported finding adaptive Parsons problems useful for learning on the end of course survey, some students with more prior programming experience had strong negative reactions to them. This implies that we should fade the scaffolding that Parsons problems provide by allowing students to choose to solve the equivalent write-code problem instead when they are presented with a Parsons problem. We are currently adding that ability to the open-source ebook platform, Runestone [103].

Students valued the intra-problem adaptation triggered by the “Help Me” button when it removed distractors or combined blocks that were not already in the correct order. Students were confused by the adaptation that provided indentation. We have modified the adaptation process to no longer provide indentation. We also modified it to select blocks to combine that are the furthest apart. We will test the effect of these changes on efficiency, learning, and student attitudes.

One of the students in the think-aloud observation used the intra-problem adaptation (“Help Me” button) to solve the first problem, but it was clear that he did not understand why a block was a distractor. This implies that we should provide a mechanism to explain what is wrong with a distractor block.

Some students reported overusing the “Help Me” button to just solve the problem to earn a point. However, a log file analysis showed that only 7.3% of the students did this. We could modify the system to not allow the “Help Me” button to be used repeatedly or perhaps not give the student credit unless they reset the problem and solve it again without help. Allowing students to choose to solve the write-code problem instead should also reduce this behavior since it was the students who said that they did not value solving Parsons problems who reported engaging in this behavior.

4.10 Conclusion

Writing code from scratch, while authentic, can overwhelm novice programmers and take an unpredictable amount of time. Students can also become stuck and unable to solve the problem without help. Adaptive Parsons problems can scaffold struggling learners and help them correctly solve problems and are usually significantly faster to solve than writing the equivalent code. However, this research suggests that if the Parsons problem solution is unusual, it may not be significantly faster to solve. Parsons problems can also be used in situations where it is hard to individually help struggling students, such as during a large lecture. However, scaffolding needs to fade as expertise develops. We plan to modify Parsons problems to truly scaffold the entire code writing process. One way that we plan to do that is to modify Parsons problems to allow students to choose to solve the equivalent write-code problem instead. In addition, we plan to try helping students who are struggling to solve a write-code problem by popping up a similar Parsons problem. Our goal is to increase the use of active learning in programming courses and to scaffold struggling students.

CHAPTER 5

Impact of Solving Parsons Problems with *(Un)*Common Solutions

To become proficient at computer programming, it is critical for novice programmers to be explicitly taught how to recognize and apply programming patterns/solutions. But how do we help them to acquire this knowledge efficiently and effectively? Chapter four revealed that an adaptive Parsons problem with an uncommon solution was not significantly more efficient to solve than writing the equivalent code. Interestingly, 77% of the students used the unusual Parsons problem solution to later solve an equivalent write-code problem. Hence, I hypothesized that changing the unusual Parsons problem solution to the most common student written solution would make that problem significantly more efficient to solve. To test the hypothesis, I conducted a mixed within/between-subjects experiment with 95 undergraduates. The results confirmed the hypothesis and its inverse. Students were significantly more efficient at solving that Parsons problem with a common solution than writing the equivalent code. Students were not significantly more efficient at solving a different Parsons problem with an uncommon solution. However, students who used the uncommon Parsons problem solution to solve an equivalent write-code problem were significantly more efficient and this resulted in higher learning gains. I also explored relationships between problem-solving efficiency, cognitive load ratings, self-efficacy, and clusters of write-code solutions. There were significant positive correlations between efficiency and cognitive load, and significant negative correlations between efficiency and self-efficacy. Previous research also found that students were confused when pressing the 'Help Me' button provided indentation and combined blocks that were already adjacent. The process was modified to no longer provide indentation and to combine blocks that were the furthest apart. To understand how students solve Parsons problems and the impact of changing the adaptation process, we report on three think-aloud observations with undergraduates. Results revealed that some students could benefit from help with planning, self-regulated learning, more explanation of distractors, and that there were no new problems due to the modifications of the adaptation process. These findings have implications for how to automatically generate and sequence adaptive Parsons problems.[1]

5.1 Introduction

Computing education theorists hypothesize that novice programmers need explicit and incremental instruction to develop at least four basic skills: code reading and tracing, code writing, pattern comprehension, and pattern application [222, 416]. Novice programmers should be taught explicitly about “stereotypical [or common] solutions to programming problems as well as strategies for coordinating and composing them”—Elliot Soloway [356, p. 850]. However, the acquisition and retention of these skills (i.e., academic growth or improvement and expertise) depends on the quality and quantity of deliberate practice [8, 106]. Time imposes limits on what can be learned and the goal is to maintain desirable difficulties while practicing programming [419].

Traditional introductory computer programming practice such as code-tracing, in which students use paper and pencil to hand trace the execution of a program [202], and code-writing, which requires students to write-code from scratch, are time-intensive, frustrating, and can decrease students’ engagement and motivation [23, 344]. Instead, drag-and-drop block-based coding exercises such as Parsons problems, also called Parsons Programming Puzzles, are increasingly being used to introduce novice computer programmers to introductory computer programming concepts more efficiently [89, 339]. Historically underrepresented minorities and females also perform better on block-based versus text-based problems [185, 407, 406]. And drag-and-drop programming platforms (also known as No-code development platforms) are also increasingly being used by companies to create programs in the absence of professional developers [415], thereby supporting the use of Parsons problems to teach novice programmers how to code.

My prior research revealed that an adaptive Parsons problem with an uncommon solution was not significantly more efficient to solve than writing the equivalent code [153]. However students who solved the Parsons problem first were more likely to use the uncommon solution when they later wrote the equivalent code [153]. Teachers and students also found parts of the original adaptation process confusing [98, 153]. In this study, we tested hypotheses based on the commonality of Parsons problems solutions and the order in which they were solved. We also explored the impact of changing the adaptation process to no longer provide indentation and to combine blocks that are the furthest apart. This has implications for how Parsons problems are generated and how we can best support adaptive learning strategies such as help-seeking via intra-problem (same problem) adaptation. It also contributes to the study of programming strategies (i.e., the common and uncommon strategies employed by novice programmers when solving programming problems and how can we increase knowledge/strategy transfer) [73, 309].

[1]Portions of this chapter have been submitted for publication.

My prior research also revealed self-reported cognitive load ratings were lower for Parsons problems than the equivalent write-code problems and that problem-solving efficiency correlated with cognitive load positively for some write-code problems [153]. Prior research shows Parsons problems not only impact cognitive but behavioral [82, 101, 283, 339] and affective [82, 98, 339] learning outcomes as well. There is evidence that self-efficacy can predict problem-solving efficiency [162]. To our knowledge, no one has explored how students' self-efficacy beliefs relate to their problem-solving efficiency during programming practice. In this study, we explored the relationship between problem-solving efficiency, cognitive load ratings, self-efficacy, and programming strategy use (i.e., commonality of solution). We can improve how adaptive Parsons problems are sequenced by analyzing the relationships between these different types of data [52, 96, 285, 382, 409]. Currently, inter-problem (between problem) adaptation only changes the difficulty of the succeeding Parsons problem based on the learners prior performance. The research questions and hypotheses were:

RQ1: What are the effects on efficiency of solving adaptive Parsons problems created from the most common student written solution or an uncommon solution versus writing the equivalent code? What are the order effects?

- **H1:** If a Parsons problem with an unusual solution is modified to use the most common student written solution then students will be more efficient at solving it.
- **H2:** If students are first presented with a Parsons problem that has an uncommon solution than a high percent will use that solution to solve an equivalent write-code problem.

RQ2: What is the effect on self-reported cognitive load ratings of solving adaptive Parsons problems versus solving equivalent write-code problems?

RQ3: How does problem-solving efficiency relate to self-reported cognitive load ratings and self-efficacy beliefs?

RQ4: Do students find the modified intra-problem (same problem) adaptation process understandable and useful?

RQ5: Why did students struggle to solve the Parsons problem with an uncommon solution?

To answer the research questions we conducted a mixed within and between-subjects experiment. We wanted to (1) investigate how common and uncommon Parsons problem solutions mediated problem-solving efficiency and (2) explore any order effects (i.e., how the order of the conditions affected students' pattern/solution acquisition). We chose a within-subjects design since problem-solving efficiency and cognitive load are dependent on the learner and thus subject to intraindividual variation [250]. We used OverCode, a system for visualising and clustering programming solutions, to determine commonality [127]. We also conducted six concurrent think-aloud observations to explore students' problem-solving behavior. We report on semi-structured interviews at the end of those observations and an end-of-course student survey to understand the impact of the changes made to the adaptation process.

This study resulted in several findings. First, students were significantly more efficient at solving Parsons problems with a common solution versus an uncommon solution. This was true of the problem we modified from our previous study [153] and every Parsons problem for which the largest cluster of students' write-code solutions matched the Parsons problem solution we used. There was a significant difference in cognitive load ratings for students who solved the modified Parsons problem first versus those who wrote the equivalent code first. Students were also significantly less efficient at solving a Parsons problem with an uncommon solution. A high percentage of students who solved the adaptive Parsons version of a problem with an uncommon solution used that solution to solve the equivalent write-code problem. This resulted in the highest learning gains. Most students who solved the write-code version of problem two first, used a different solution than the adaptive Parsons problem. There were two clusters of the same size, but with very different solutions for this problem. Self-efficacy beliefs correlated negatively with efficiency when solving Parsons problems with both a common or uncommon solution. Overall, students were significantly more efficient at solving write-code problems after they solved an equivalent Parsons problem regardless of whether or not the solution was common. The highest learning gains for solving a write-code problem after solving an equivalent Parsons problem with a common solution was for the second attempt at the hardest write-code problem.

Second, the think-aloud observations revealed students may struggle to solve Parsons problems with uncommon solutions because they need help with planning, self-regulated learning, and distractor blocks. Semi-structured interviews did not uncover any problems with the changes we made to the Parsons problem adaptation process. The end-of-course student survey provided evidence that more students found the adaptation useful and more understood the process.

5.2 Methods

To answer the research questions we conducted a (1) mixed within and between-subjects experiment (2) a concurrent think-aloud study with open-ended questions at the end of each session, and (3) and an end-of-course student survey. This study can be categorize as a constructive replication which "refers to the effort to test prior findings using different experimental designs, measurements, and data analysis techniques that are more robust than prior studies" [see 144, p. 42:3].

5.2.1 Participants

We received institutional review board (IRB) approval to recruit participants from a post-secondary research institution in the northern Midwest in the United States. The participants were all enrolled in a data-oriented programming course in Python during the winter semester (between January and April) of 2021 (N = 144). This course is the second Python course for School of Information majors, though other majors take it as well. It requires prior programming experience. The course focuses on developing intermediate programming skills in Python and covers working with data from a variety of sources (strings, files, APIs, websites, and databases), object-oriented programming basics, regular expressions, debugging, testing, and SQL. Fifty-three percent of the students identified as female and 47% identified as male; 33% percent identified as Asian, 2% as Black, 6% as Hispanic, 46% as White, 4% as Multiracial, and 9% did not indicate their race. The ages ranged from 18 to 33 years old (M = 20 years old, SD = 1.49). Eleven percent were Computer Science (CS) majors, 3% were Data Science (DS) majors, 3% were Information Science (IS) majors, and 83% were majors in other disciplines. The average maximum American College Test (ACT) math score was 31 (SD = 3.42) on a scale ranging from 1 (low) to 36 (high). The average GPA was 3.721 (SD = 0.448).

5.2.2 Materials

We used the Introductory Programming Self-Efficacy Scale (IPSES) [2] [361] to measure students' beliefs about introductory computer programming concepts and competences. The scale has 20 items that comprise four factors: tracing program flow (Factor 1); controlling program flow (Factor 2); using structures and patterns for problem-solving (Factor 3); and persistence, debugging, and problem-solving competences (Factor 4). It asks respondents to rate their confidence in doing tasks related to these four factors using a 7-point Likert scale from "strongly disagree" to "strongly agree" and "no answer" if a specific term or task is totally unfamiliar to the respondent. We administered the scale at the beginning of the semester (January 27th, week 2) and at the end of the semester (April 19th, week 14). We obtained a total of 143 responses (a 99% response rate) at the beginning and 110 responses (a 76% response rate) at the end (107 were repeat respondents). We calculated reliability using R's psych package [305] and report both Cronbach's α and McDonald's ω for the both administrations of the scale given methodological disputes [see 151, 426]. We decided to present this because the scale is new and our results add to its validity. Cronbach's α for the scale was 0.95 and 0.96 respectively; this indicates high test-retest reliability. McDonald's

[2] https://go.wwu.de/qpuoe

ω for the scale was 0.72 and 0.73 respectively. The alpha reliabilities of the scores on the four factors were... tracing program flow (Factor 1) = 0.93, controlling program flow (Factor 2) = 0.92, using structures and patterns for problem-solving (Factor 3) = 0.90, and persistence, debugging, and problem-solving competences (Factor 4) = 0.90. These high reliabilities are consistent with Steinhorst et al.'s [361] with the exception that Factor 4 was lower than 0.95 as recommended [376].

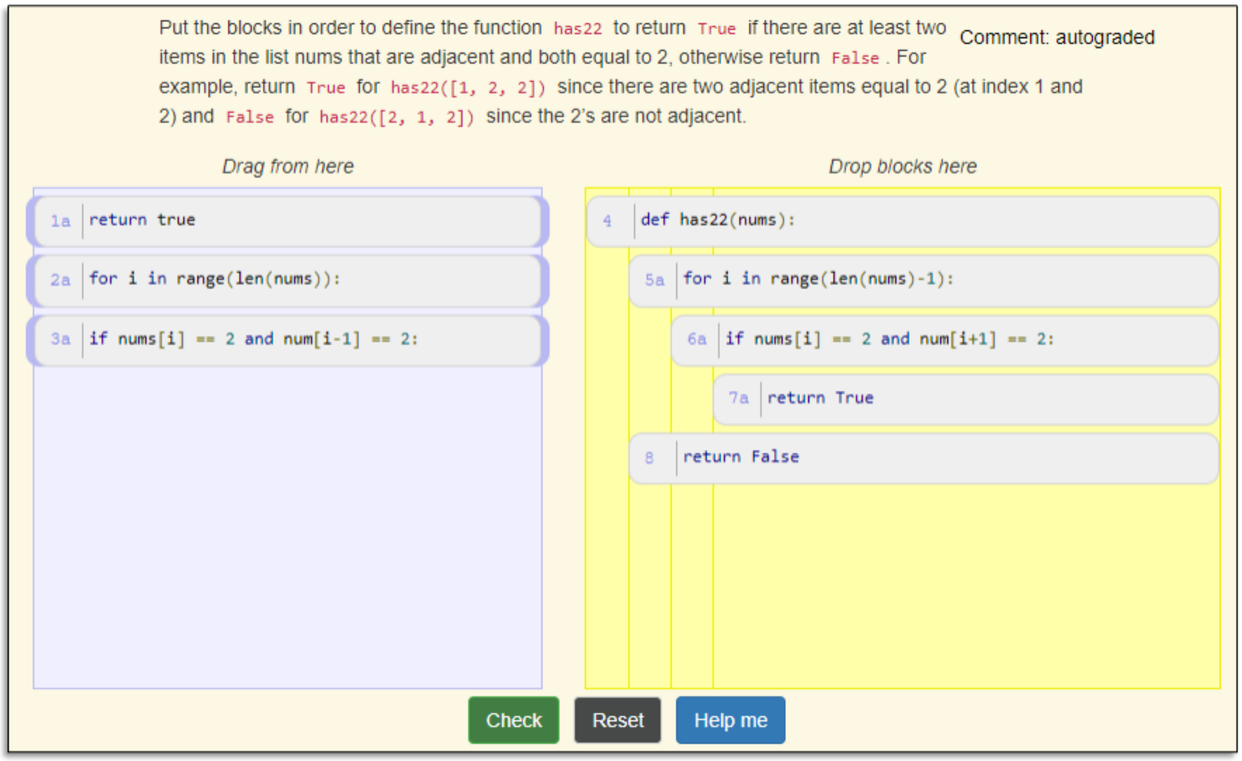
We used two versions of a problem set from our previous study [153] with the exception of changing one of the Parsons problem solutions which was unusual to match the most common student written solution; this change is shown in Figure 5.1. The only difference between version A and B was the problem type. The second problem in version A was a write-code problem as shown in Figure 5.2. In version B, the same problem was presented as a Parsons problem as shown in Figure 5.3. Each version included five problems in the order shown in Table 5.1. To alter the amount of expected cognitive load needed to solve each problem, the problems ranged in difficulty from easy to hard. The concepts covered include: strings, lists, ranges, conditionals, loops, dictionaries, and functions. Some of the problems came from past Advanced Placement (AP) Computer Science (CS) A exams and some from the CodingBat website created by Nick Parlante of Stanford University [282]; they exemplify problems covered in introductory computer programming courses. The problems are available as requested.

Table 5.1: Order of Problem Type by Version

Version	Problem Type
A	Parsons*, Write, Parsons, Write, Parsons
B	Write, Parsons, Write, Parsons, Write

Note: Asterisks indicate a change in the solution. Each percentage represents those who got it correct out of the n for each problem in Table 5.4.

The Pass scale was administered after each problem in both versions [271]. This question uses a 9-point Likert scale that asks respondents to rate how much mental effort they invested in solving the previous problem from *"very, very low mental effort"* to *"very, very high mental effort"* as shown in Additional Material 1.2.


Put the blocks in order to define the function `has22` to return `True` if there are at least two items in the list nums that are adjacent and both equal to 2, otherwise return `False`. For example, return `True` for `has22([1, 2, 2])` since there are two adjacent items equal to 2 (at index 1 and 2) and `False` for `has22([2, 1, 2])` since the 2's are not adjacent.

Comment: autograded

Drag from here

```
1a return true
2a for i in range(len(nums)):
3a if nums[i] == 2 and num[i-1] == 2:
```

Drop blocks here

```
4  def has22(nums):
5a     for i in range(len(nums)-1):
6a         if nums[i] == 2 and num[i+1] == 2:
7a             return True
8      return False
```

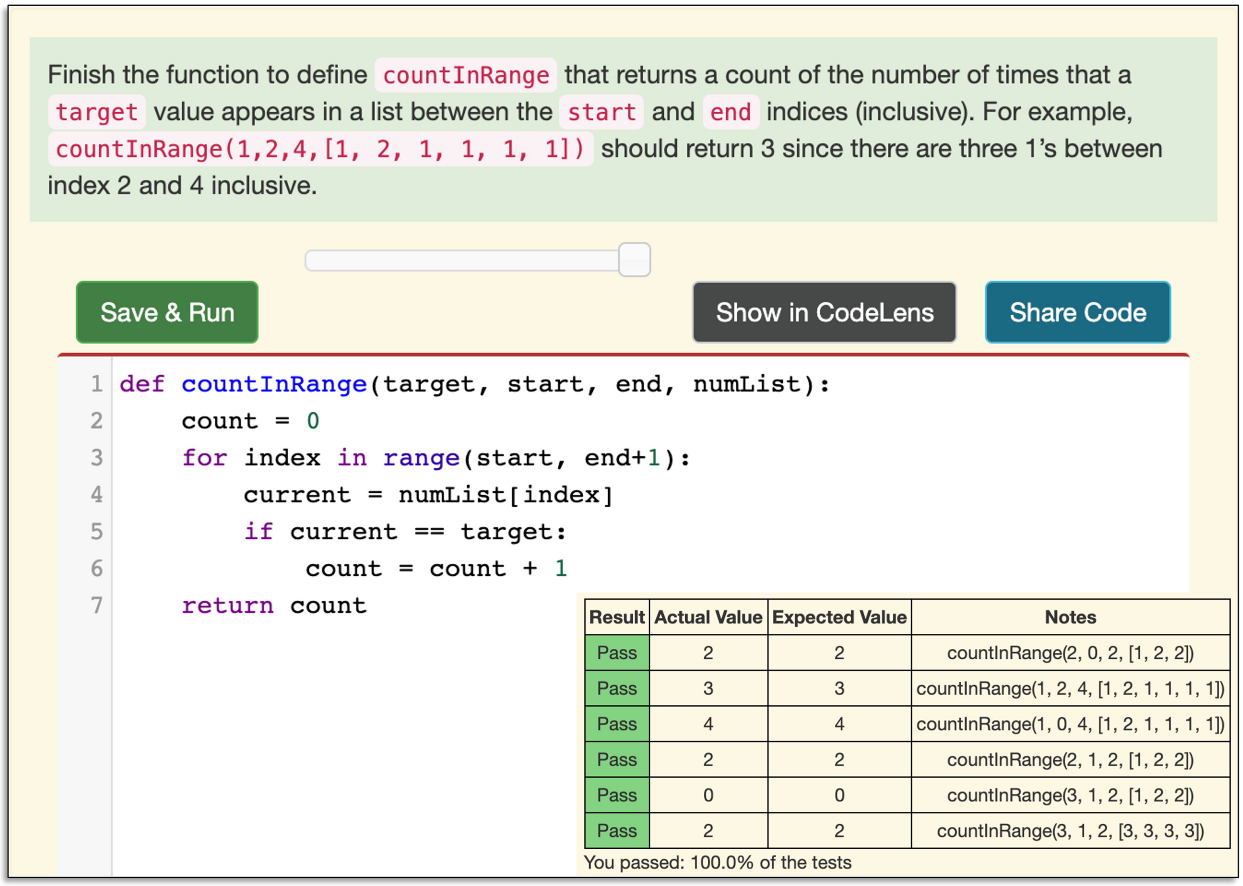
Figure 5.1: *First problem in Version A as an adaptive Parsons problem (Problem 1 in Table 5.4).*

5.3 Mixed Within- and Between-Subjects Experiment

5.3.1 Experimental Design

We conducted a mixed within and between-subjects experiment [55] to (1) test the hypothesis that students would be more efficient at solving adaptive Parsons problems with common solutions than writing the equivalent code, (2) explore how the order of completing each problem type affected students' subsequent problem-solving behavior (i.e., efficiency and the acquisition of a pattern/solution), and (3) explore the relationships between efficiency and self-reported cognitive load ratings and efficiency and self-efficacy beliefs.

The first part of the experiment was started on Feb 16th; students were randomly assigned to one version of the problem set (A or B). The second part was due March 10th; students completed the opposite version of the problem set (A or B). Students could earned 10 extra lecture participation points for completing each version of the problem set and were asked to work individually. At the end of the course (week 15), participants were asked to complete both versions (A and B) of the problem set as an extra credit assignment. Students needed to earn 2,000 points or more in during the course to receive an A+.



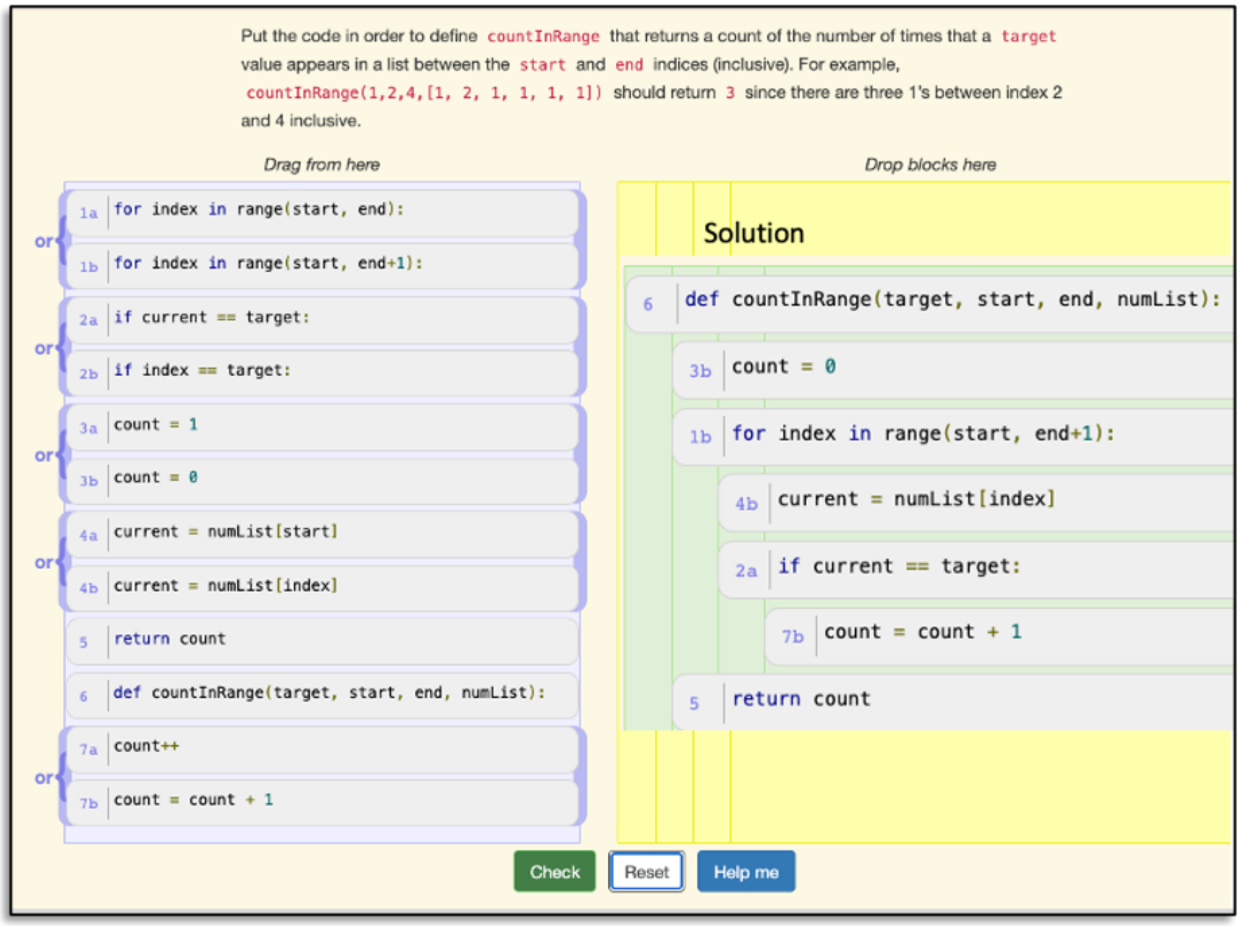
Finish the function to define `countInRange` that returns a count of the number of times that a `target` value appears in a list between the `start` and `end` indices (inclusive). For example, `countInRange(1,2,4,[1, 2, 1, 1, 1, 1])` should return 3 since there are three 1's between index 2 and 4 inclusive.

```python
def countInRange(target, start, end, numList):
    count = 0
    for index in range(start, end+1):
        current = numList[index]
        if current == target:
            count = count + 1
    return count
```

Result	Actual Value	Expected Value	Notes
Pass	2	2	countInRange(2, 0, 2, [1, 2, 2])
Pass	3	3	countInRange(1, 2, 4, [1, 2, 1, 1, 1, 1])
Pass	4	4	countInRange(1, 0, 4, [1, 2, 1, 1, 1, 1])
Pass	2	2	countInRange(2, 1, 2, [1, 2, 2])
Pass	0	0	countInRange(3, 1, 2, [1, 2, 2])
Pass	2	2	countInRange(3, 1, 2, [3, 3, 3, 3])

You passed: 100.0% of the tests

Figure 5.2: *Second problem in Version A as a write-code problem (Problem 2 in Table 5.6).*



Put the code in order to define `countInRange` that returns a count of the number of times that a `target` value appears in a list between the `start` and `end` indices (inclusive). For example, `countInRange(1,2,4,[1, 2, 1, 1, 1, 1])` should return `3` since there are three 1's between index 2 and 4 inclusive.

Drag from here

```
1a  for index in range(start, end):
or
1b  for index in range(start, end+1):

2a  if current == target:
or
2b  if index == target:

3a  count = 1
or
3b  count = 0

4a  current = numList[start]
or
4b  current = numList[index]

5   return count

6   def countInRange(target, start, end, numList):

7a  count++
or
7b  count = count + 1
```

Drop blocks here

Solution

```
6   def countInRange(target, start, end, numList):
3b      count = 0
1b      for index in range(start, end+1):
4b          current = numList[index]
2a          if current == target:
7b              count = count + 1
5       return count
```

Figure 5.3: *Second problem in Version B as an adaptive Parsons problem (Problem 2 in Table 5.4).*

5.3.2 Participants

In this section, we report on data from the participants (*n* = 95) who completed both the adaptive Parsons version and equivalent code writing version for some or all of the problems correctly; this ranged from 26 to 62 as shown in Tables 5.4 and 5.6.

5.3.3 Analysis

Task completion times were calculated for each problem using a Python script for adaptive Parsons problems (timeCorrectParsons.py) and a separate one for equivalent write-code problems (timeCorrectWriteCode.py). These Python scripts can be used and modified by other researchers who run experiments using Runestone'sANON's free interactive computing education eBooks [10] to study and improve time-on-task metrics for programming practice such as [214].

Learning gains researchers posit one of the limitations to using pre-post test is using the same test for both stages and express learning as a "transitional experience" that should involve more authentic ways of assessment [314, p. 22]. Hence, solving adaptive Parsons problems followed by equivalent/isomorphic write-code problems is one way to transition to the authentic task of writing code from scratch. In addition, the fragile nature of learning in introductory computer programming courses [310] is yet another reason to consider using both novice-friendly (Parsons problems) and more authentic forms of assessment such as write-code problems to measure learning gains. Computing education researchers who've performed multi-national studies interpret students' performance on tasks designed to assess basic programming skills as suggesting that many "students have a fragile grasp of skills that are a prerequisite for problem-solving" upon completing CS1 courses [221]. Furthermore, prior research in computers and physics education has shown that paired isomorphic multiple-choice questions can be used to study learning gains [251]. Paired problems are isomorphic if they require learners to solve them using the same principle [351].

We used the percentage of correct solutions for up to three attempts of solving a Parsons problems and then the equivalent write-code problem to compute raw and normalized learning gains. This is based on prior work using isomorphic problems and calculating normalized learning gains to account for ceiling effects [190]. Here are the equations:

$$Raw\ Learning\ Gain = (\%Correct\ WriteCode\ Problem) - (\%Correct\ Parsons\ Problem) \tag{5.1}$$

$$Normalized\ Learning\ Gain = \frac{(\%Correct\ WriteCode\ Problem) - (\%Correct\ Parsons\ Problem)}{(100\% - (\%Correct\ Parsons\ Problem)} \tag{5.2}$$

Statistical analysis was performed using RStudio [377]. We ran Wilcoxon Matched-Pairs Signed-Ranks test to analyze the difference in the median times to the first correct solution within the groups since this was a within-subjects study and the data violated assumptions of normality and equal variances [260]. We also report the mean and standard deviation for these differences based on [229]. To do this, we used the 'wilcox.test' function from of the stats R package. We also ran Mann–Whitney U tests between groups to explore order effects for certain problems. Finally, we (1) ran paired *t*-test on cognitive load ratings to analyze the difference between problem types [164], (2) computed probability-based effect sizes [66, 323] using the canprot R package [83] and Cohen's d_{rm} using the R package lsr [264], (3) adjusted for any multiple comparisons were using Bonferroni's correction [59], and (4) ran Spearman correlation coefficient tests [149] to analyze the relationships between efficiency and self-reported cognitive load ratings and efficiency and self-efficacy.

To analyze the Introductory Programming Self-Efficacy Scale (IPSES), we used the alpha and omega function from the psych R package to calculate reliability and the cluster, factoextra, and fpc packages to perform k-means cluster analysis. We chose the elbow method [31] to determine clusters and arrived at four groups (see Figure 5.4 and Table 5.2). Computing education researchers investigating computer programming self-efficacy have arrived at four distinct groups as far back as [299] and as recent as [188].

Table 5.2: Self-Efficacy Clusters

Cluster	*n*	Factor 1	Factor 2	Factor 3	Factor 4
1 Low	39	5.709	6.060	2.651	4.855
2 Low Average	26	4.019	4.051	2.915	3.410
3 Average High	43	5.674	6.101	4.600	5.333
4 High	35	6.738	6.829	6.200	6.233

Notes: Factor 1 = Tracing program flow; Factor 2 = Controlling program flow; Factor 3 = Using structures and patterns for problem-solving; Factor 4 = Persistence, debugging, and problem-solving competences.

5.3.4 Results and Discussion

First we present both the within and between-subject study results to answer RQ1. Then we discuss implications and future work related to this question. Second, we explain how the within-subject study results answer RQ2 and RQ3. Then we discuss implications and future work related to these questions.

The task completion times for students who solved the adaptive Parsons problem version of a

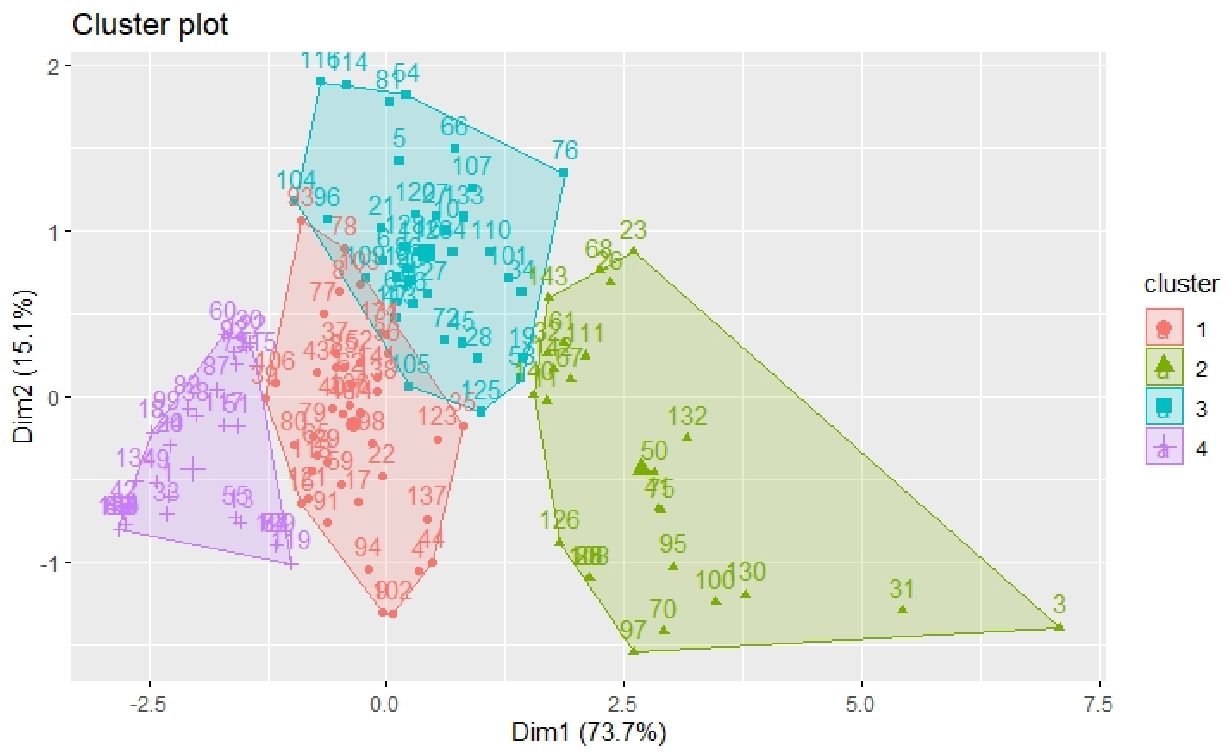


Figure 5.4: *K-means cluster analysis*

problem before solving the equivalent write-code version are shown in Table 5.4; the self-reported cognitive load ratings are shown in Table 5.8. The task completion times for students who solved the write-code version of a problem before solving the equivalent adaptive Parsons problem version are shown in Table 5.6; the self-reported cognitive load ratings are shown in Table 5.9. We used OverCode to confirm whether or not the solutions to the Parsons problems matched the most common student written solutions; this is denoted by the equivalent symbol ($^{\equiv}$). Problem one (has22), which we changed based on our previous study to represent the most common student written solution, remained the most common. OverCode also confirmed that all of the Parsons problem solutions we presented to students matched the most common student written solutions except for problem two (countInRange). Students who solved the write-code version of problem two (see Figure 2.2) before solving the equivalent Parsons problem used solutions that were different from the provided Parsons problem solution (see Figure 5.3). This problem asked students to finish creating a function that returned the number of times a specific number appeared in a list between the start and end indices (see Figure 5.2 and Appendix B for details about the commonality of student written solutions).

Learning gains are shown in Table 5.3. Using the equations in the section above, we provide an example. Of the 29 students who did problem two (see table 5.4), 23% got the Parsons problem correct in one attempt and then of the 29 students, 61% solved the write-code problem correct in one attempt. This problem, which presented students with an uncommon Parsons problem solution, resulted in the highest raw learning gain of 38% and normalized learning gain of 49%. The highest learning gains for solving a write-code problem after solving an equivalent Parsons problem with a *common* solution was for the second attempt at the hardest write-code problem five.

Table 5.3: Learning Gains for Parsons → Write

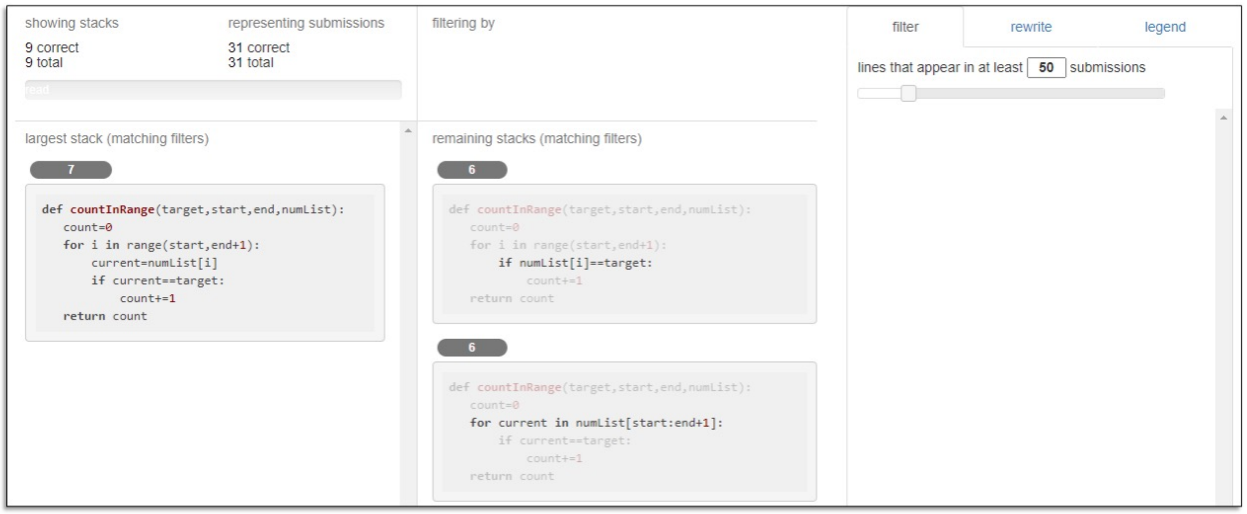
	Learning gain			Normalized learning gain		
	Attempts			Attempts		
Problem	1	2	3	1	2	3
1 has22 (H)	-5%	0%	-2%	-8%	0	-2%
2 countInRange (M)	38%	-16%	10%	49%	-23%	10%
3 diffMaxMin (E)	-18%	5%	5%	-225%	5%	5%
4 dictTotal (M)	-12%	6%	0%	-55%	7%	0%
5 dictNames (H)	-24%	11%	5%	-42%	14%	5%

Notes: Attempts represent submissions before correctly solving the problem.

5.3.4.1 *RQ1: What are the effects on efficiency of solving an adaptive Parsons problems made with the most common or uncommon student written solution versus writing the equivalent code?*

The results supported **H1: Students were significantly more efficient at solving a Parsons problem we modified to use the most common student written solution instead of the uncommon solution we used in our previous study versus solving the equivalent write-code problem.** The median time to solve each Parsons problem was significantly less than the median time to write the equivalent code for all of the problems and both versions except for problem two of version B (see Figure 5.3). It was intended to be of average difficulty. It took students significantly more time to solve problem two as a Parsons problem before solving it as a write-code problem (see Table 5.4). The Parsons problem solution used in problem two was uncommon. This supports the inverse hypothesis that: **Students were significantly less efficient at solving a Parsons problem with an uncommon solution than solving the equivalent write-code problem.**

The results also supported **H2: Students first presented with a Parsons problem that had an uncommon solution tended to use that solution to solve the equivalent write-code problem.** The largest OverCode cluster of solutions for which students solved problem two as an equivalent write-code problem after the Parsons problem showed that these students used the Parsons problem solution to solve the write code problem (see Figure 5.5). They did this with the exception of


showing stacks
9 correct
9 total
representing submissions
31 correct
31 total
filtering by
filter
rewrite
legend
lines that appear in at least 50 submissions
largest stack (matching filters)
remaining stacks (matching filters)
7
6
6

```
def countInRange(target,start,end,numList):
   count=0
   for i in range(start,end+1):
       current=numList[i]
       if current==target:
           count+=1
   return count
```

```
def countInRange(target,start,end,numList):
   count=0
   for i in range(start,end+1):
       if numList[i]==target:
           count+=1
   return count
```

```
def countInRange(target,start,end,numList):
   count=0
   for current in numList[start:end+1]:
       if current==target:
           count+=1
   return count
```

Figure 5.5: *OverCode visualization of solutions for those who solved Problem 2 in Table 5.4 as Parsons problem first. The top-left panel displays the number of clusters (9), called stacks, and the total number of visualized solutions (31). The panel below this in the first column shows the largest stack which comprises 7 solutions. The second column displays the remaining stacks. The third column displays the lines of code occurring in the cleaned solutions of the stacks together with their frequencies [127].*

writing **`count += 1`** instead of **`count = count + 1`**. Whether the Parsons problem solution was common or uncommon, the largest cluster of students who solved the Parsons version first used that solution to solve the equivalent write-code problem.

Table 5.4: Task Completion Times for Parsons → Write

		Parsons Problem	Write-Code Problem	Wilcoxon Matched-Pairs Signed-Ranks Test		
Problem (Diff.)	*n*	*Mdn* in seconds	*Mdn* in seconds	*V*	*p*-value	*A*
1 has22 (H)$^{\equiv}$	57	50.0	96.0	279.0	$p < 0.001$***	0.65
2 countInRange (M)$^{\equiv}$	29	125.0	68.0	325.5	$p = 0.020$*	0.33
3 diffMaxMin (E)$^{\equiv}$	59	11.0	37.0	87.5	$p < 0.001$***	0.65
4 dictTotal (M)$^{\equiv}$	30	24.0	30.0	89.5	$p = 0.006$**	0.69
5 dictNames (H)$^{\equiv}$	50	82.0	186.0	174.5	$p < 0.001$***	0.68

Notes: E = Easy, M = Medium, H = Hard; The equivalent symbol ≡ indicates that students who solved the adaptive Parsons problem first used the same solution to solve the equivalent write-code problem; * p < .05, ** p < .01, *** p < .001; *A* = probability-based effect size measure (nonparametric generalization of common language effect size statistic).

We conducted Mann–Whitney U tests to determine whether there was a significant difference in efficiency between the order in which the two groups solved problem one (has22) and problem two (countInRange). Results showed a significant difference between the adaptive Parsons problems for problem one (has22) (W = 1140, p-value = 0.01105) and problem two (countInRange) (W = 623.5, p-value = 0.02448) due to order effects. **Students were significantly more efficient**

Table 5.5: Average Task Completion Times for Parsons $\rightarrow$ Write

		Parsons Problem	Write-Code Problem
Problem (Diff.)	*n*	*M (SD)* in seconds	*M (SD)* in seconds
1 has22 (H)$^{\equiv}$	57	73.51 (72.95)	194.14 (297.57)
2 countInRange (M)$^{\equiv}$	29	137.14 (72.02)	93.31 (72.34)
3 diffMaxMin (E)$^{\equiv}$	59	14.58 (12.53)	76.61 (158.41)
4 dictTotal (M)$^{\equiv}$	30	23.50 (6.50)	49.50 (50.88)
5 dictNames (H)$^{\equiv}$	50	117.32 (119.67)	350.18 (481.65)

Notes: E = Easy, M = Medium, H = Hard; The equivalent symbol $\equiv$ indicates that students who solved the adaptive Parsons problem first used the same solution to solve the equivalent write-code problem.

Table 5.6: Task Completion Times for Write $\rightarrow$ Parsons

		Parsons Problem	Write-Code Problem	Wilcoxon Matched-Pairs Signed-Ranks Test		
Problem (Diff.)	*n*	*Mdn* in seconds	*Mdn* in seconds	*V*	*p*-value	*A*
1 has22 (H)$^{\equiv}$	30	39.5	205.0	5.0	$p < 0.001$***	0.77
2 countInRange (M)	61	89.0	123.0	501.5	$p = 0.002$**	0.65
3 diffMaxMin (E)$^{\equiv}$	30	10.0	128.5	0	$p < 0.001$***	0.84
4 dictTotal (M)$^{\equiv}$	62	25.0	56.5	163.0	$p < 0.001$***	0.66
5 dictNames (H)$^{\equiv}$	26	54.5	388.0	14.0	$p < 0.001$***	0.82

Notes: E = Easy, M = Medium, H = Hard; The equivalent symbol $\equiv$ indicates that students who solved the write the code problem first used the same solution as the adaptvie Parsons problem; * $p < .05$, ** $p < .01$, *** $p < .001$; *A* = probability-based effect size measure (nonparametric generalization of common language effect size statistic).

at solving a Parsons problem with either a common (has22) or uncommon (countInRange) solution when they solved an equivalent write-code problem first than students who did the opposite. Results also showed a significant difference between the write-code problems for problem one (has22) (W = 581.5, p-value = 0.01477) and problem two (countInRange) (W = 1239.5, p-value = 0.002207) due to order effects. **Students were significantly more efficient at solving a write-code problem when they solved a Parsons problem with either a common (has22) or uncommon (countInRange) solution first than students who did the opposite.**

In summary, the results show that if you solved the write-code version of problem one (common solution) and two (uncommon solution) and then solved the Parsons version, you were significantly more efficient at solving the Parsons version. In addition, if you solved the Parsons problem version of problem one (common solution) and two (uncommon solution) and then solved the write-code version you were significantly more efficient at solving the write-code version.

The probability-based measure of effect sizes (*A*) shown in Table 5.4 mean that the probability ranges from 33% to 69% that a randomly selected student will be significantly more efficient at

Table 5.7: Average Task Completion Times for Write $\rightarrow$ Parsons

		Parsons Problem	Write-Code Problem
Problem (Diff.)	*n*	*M (SD)* in seconds	*M (SD)* in seconds
1 has22 (H)$^{\equiv}$	30	40.47 (18.93)	379.47 (465.11)
2 countInRange (M)$^{\equiv}$	61	110.46 (71.80)	257.20 (383.89)
3 diffMaxMin (E)$^{\equiv}$	30	11.40 (7.75)	215.07 (203.98)
4 dictTotal (M)$^{\equiv}$	62	29.11 (24.79)	97.44 (166.43)
5 dictNames (H)$^{\equiv}$	26	58.69 (25.89)	476.77 (455.31)

Notes: E = Easy, M = Medium, H = Hard; The equivalent symbol $\equiv$ indicates that students who solved the adaptive Parsons problem first used the same solution to solve the equivalent write-code problem.

solving a Parsons problem with a common solution than writing the equivalent code when they complete the Parsons problem first. The probability-based measure of effect sizes (*A*) shown in Table 5.6 mean that the probability ranges from 65% to 84% that a randomly selected student will be significantly more efficient at solving a Parsons problem with a common or uncommon solution than writing the equivalent code when they complete the latter first.

To be more efficient than writing the equivalent code, Parsons problems should be generated from the most common student written code. We plan to mine the huge amount of student written code from free and interactive eBooks on the Runestone eBook platform to automatically generate Parsons problems by using OverCode to determine the most common student solution to write-code problems. The notion of common (stereotypical) solutions/plans has also been used to create grading schemes for write-code assignments and researchers suggest this kind of evaluation can be used to increase instructors' awareness of students' difficulties with programming tasks [69]. We plan to explore how uncommon solutions relate to difficulties with programming practice and learning gains. Students also recall and use Parsons problem solutions to solve equivalent write-code problems. If students are more efficient at solving a write-code problem after solving a Parsons problem with a common solution, there may be no learning gains because they have advanced beyond novice for the concepts that problem addresses or have acquired that schema. This type of information can help with placement and providing students with challenges. In contrast, if students are more efficient at solving a write-code problem after solving a Parsons problem with an uncommon solution, there may be more learning gains because they had not advanced beyond novice for the concepts that problem addressed and had not yet acquired that schema. Furthermore, the between-subjects analysis of problem one and problem two provided evidence that whether or not the Parsons problem solution is common or uncommon, students are significantly more efficient at solving the adaptive Parsons version after solving the equivalent write-code problem. What is the relationship between common and uncommon solutions and desirable difficulties? Desirable ifficulties "slow down the acquisition rate of learning, but facilitate long-term retention and transfer [57, p. 1]. And, what about problem order (interleaved or blocked)? This also affects learning transfer [see 217]. Learners may benefit form interleaved problem order when learning *to apply* a skill and from blocked problem order when learning *how to apply* a skill [217]. We plan to explore how uncommon solutions can be used to alert instructors too potential knowledge gaps and opportunities for learning transfer. Future research should also experiment with different measures of efficiency as mentioned in the related work section (see Efficiency and Time-on-Task).

5.3.4.2 *RQ2: What is the effect on self-reported cognitive load ratings of solving adaptive Parsons problems versus solving equivalent write-code problems?*

The results are shown in Table 5.8 and Table 5.9. **Students reported investing significantly less mental effort in solving problem one as a Parsons problem than the equivalent write-code problem.** This was the problem for which we changed the solution from our previous study. Even though insignificant, students invested more mental effort in solving a Parsons problem with an uncommon solution (problem two) then the equivalent write-code problem when they solved the Parsons problem first. But when they solved the write-code version of problem two first, they reported investing lower mental effort in solving the equivalent Parsons problem even though the solution was uncommon. Solving a Parsons problem with an uncommon solution before solving an equivalent write-code problem resulted in slightly lower self-reported cognitive load ratings. Students also reported investing significantly less mental effort in solving problem three—the easiest problem—as a Parsons problem regardless of the order in which they completed it.

Table 5.8: Cognitive Load Ratings for Parsons $\rightarrow$ Write

	Parsons Problem	Write-code Problem	Paired *t* test				
Problem (Diff.)	*M (SD)* of ratings	*M (SD)* of ratings	*t* value	*df*	*p*-value	Cohen's d_{rm}	*A*
1 has22 (H)	2.60 (1.73)	3.28 (2.19)	-2.5926	56	$p = 0.012$*	0.34	0.60
2 countInRange (M)	3.86 (1.71)	3.34 (1.99)	1.4648	28	$p = 0.154$	-0.27	0.42
3 diffMaxMin (E)	1.17 (1.52)	2.42 (1.83)	-5.3916	58	$p < 0.001$***	0.73	0.70
4 dictTotal (M)	1.93 (1.39)	2.43 (1.93)	-1.4936	29	$p = 0.146$	0.28	0.58
5 dictNames (H)	3.42 (1.72)	3.34 (2.36)	0.20216	49	$p = 0.841$	-0.04	0.49

Notes: E = Easy, M = Medium, H = Hard; * $p < .05$, ** $p < .01$, *** $p < .001$, paired t-test; Likert scale: 1 = Very, very low mental effort; 1 = Very low mental effort; 3 = Low mental effort; 4 = Rather low mental effort, 5 = Neither low nor high mental effort; 6 = Rather high mental effort; 7 = High mental effort; 8 = Very high mental effort, 9 = Very, very high mental effort

There was a significant difference in cognitive load ratings for students who solved the modified Parsons problem first versus those who wrote the equivalent code first. Prior work showed that students self-reported investing less mental effort in solving Parsons problems than isomorphic write-code problems without looking at order effects [153]. We analyzed the data to draw

Table 5.9: Cognitive Load Ratings for Write $\rightarrow$ Parsons

	Parsons Problem	Write-code Problem	Paired *t* test				
Problem (Diff.)	*M (SD)* of ratings	*M (SD)* of ratings	*t* value	*df*	*p*-value	Cohen's d_{rm}	*A*
1 has22 (H)	2.57 (1.74)	3.30 (2.28)	-1.8422	29	$p = 0.076$	0.35	0.60
2 countInRange (M)	3.11 (2.03)	3.39 (2.24)	-1.2159	60	$p = 0.229$	0.13	0.54
3 diffMaxMin (E)	1.00 (1.17)	2.77 (1.85)	-5.7071	29	$p < 0.001$***	1.07	0.79
4 dictTotal (M)	2.15 (1.64)	2.56 (1.90)	-1.7266	61	$p = 0.089$	0.23	0.57
5 dictNames (H)	3.04 (1.80)	3.00 (2.90)	0.058921	25	$p = 0.954$	-0.02	0.50

Notes: E = Easy, M = Medium, H = Hard; * $p < .05$, ** $p < .01$, *** $p < .001$, paired t-test; Likert scale: 1 = Very, very low mental effort; 1 = Very low mental effort; 3 = Low mental effort; 4 = Rather low mental effort, 5 = Neither low nor high mental effort; 6 = Rather high mental effort; 7 = High mental effort; 8 = Very high mental effort, 9 = Very, very high mental effort

out order effects and found that, in general, students self-reported investing more mental effort in solving write-code problems versus Parsons problems except for problem two (medium) and five (hard) across both versions of the problem set. This provides evidence that at least some students perceive these two problems as more difficult tasks that require more mental effort, but the difference was not significant. This may be because cognitive load and task difficulty/complexity are also related to interest. Researchers posit that "conceptually, a task with high cognitive load should be a task that is perceived to be difficult and cannot be done well even though the individual likes the task and invests a high level of effort in it" [420, p. 3]. Learners who are not interested or do not want to well on a difficult task may give up instead of investing more effort. This would lead to longer completion times that effect the results of statistical tests. In contrast, the difference in self-reported cognitive load was significant across both versions of the problem set for problem three, which was the easiest. Is it easier to self-assess the mental effort involved in solving easy tasks? Patently low complexity tasks improve our ability to track variations in a learner's response to task complexity [117]. These results provide evidence that an easy problem requires significantly more mental effort to solve as a write-code problem than a Parsons problem independent of the order in which you solve it. Task cognitive load and task difficulty can also be use in conjunction with other programming task variables for problem sequencing and student modeling; perhaps we need to measure interest given its relationship to task difficulty [96]. Future research should investigate other hypothesis related to modifying the type and amount of information for programming problems such as the Trade-off Hypothesis [353] and Cognition Hypothesis [313]. These stem from research on second language learning which parallels learning a programming language [277, 312]. Future research should also investigate creating more interest-driven programming practice assignments [11, 173].

5.3.4.3 *RQ3: How does problem-solving efficiency relate to self-reported cognitive load ratings and self-efficacy beliefs?*

The results for the relationship between problem-solving efficiency and self-reported cognitive load ratings are shown in Table 5.10 and 5.11. The strongest significant positive correlation was for problem five when solved as a Parsons problem first. This was indeed the hardest problem; it had the second most blocks after problem two, but its solution was common. There was a significant positive correlation for problem two when solved as a Parsons problem second. This was the only problem with an uncommon solution. **Students' problem-solving efficiency significantly positively correlated with their self-reported cognitive load ratings when solving a Parsons problem with a common solution before an equivalent write-code problem. Students' problem-solving efficiency significantly positively correlated with their self-reported cognitive load ratings when solving a Parsons problem with an uncommon solution after solving the equivalent write-code problem.** The strongest significant positive correlations for write-code problems were for problem three and four when solved as a write-code problem first.

Table 5.10: Correlations between Task Completion Times and Cognitive Load Ratings for Parsons $\rightarrow$ Write

		Parsons Problem		Write-code Problem	
Problem	*n*	ρ	*p*-value	ρ	*p*-value
1 has22 (H)	57	.13	0.325	.01	0.925
2 countInRange (M)	29	.17	0.372	.03	0.866
3 diffMaxMin (E)	59	.09	0.488	.20	0.130
4 dictTotal (M)	30	.22	0.239	.03	0.870
5 dictNames (H)	50	.40	0.004**	-.16	0.270

Notes: * p < .05, ** p < .01, *** p < .001, ρ = Spearman rank-order correlation coefficient; Likert scale: 1 = Very, very low mental effort; 1 = Very low mental effort; 3 = Low mental effort; 4 = Rather low mental effort, 5 = Neither low nor high mental effort; 6 = Rather high mental effort; 7 = High mental effort; 8 = Very high mental effort, 9 = Very, very high mental effort

The results for relationships between problem-solving efficiency and self-efficacy beliefs are shown in Table 5.12. There was a significant negative relationship for two Parsons problems (one and two) and one write code problem (four). None of the problems showed a significant relationship across problem type even though they were isomorphic. **Students' problem-solving efficiency significantly positively correlated with their self-efficacy beliefs when solving Parsons problems with either a common or uncommon solution.**

Table 5.11: Correlations between Task Completion Times and Cognitive Load Ratings for Write $\rightarrow$ Parsons

		Parsons Problem		Write-code Problem	
Problem	*n*	ρ	*p*-value	ρ	*p*-value
1 has22 (H)	30	-.15	0.436	-.06	0.754
2 countInRange (M)	61	.35	0.006**	-.05	0.720
3 diffMaxMin (E)	30	.24	0.200	.58	0.001***
4 dictTotal (M)	62	.09	0.487	.30	0.017*
5 dictNames (H)	26	-.03	0.898	-.33	0.099

Notes: * p < .05, ** p < .01, *** p < .001, ρ = Spearman rank-order correlation coefficient; Likert scale: 1 = Very, very low mental effort; 1 = Very low mental effort; 3 = Low mental effort; 4 = Rather low mental effort, 5 = Neither low nor high mental effort; 6 = Rather high mental effort; 7 = High mental effort; 8 = Very high mental effort, 9 = Very, very high mental effort

There were two significant relationships between problem-solving efficiency and cognitive load for Parsons problems. Prior work showed a strong significant positive relationship for problem one and two as write-code problems without looking at order effects [153]. Our results revealed a significant positive relationship for problem two (with an uncommon solution) when solved as a Parsons problem second and an insignificant, yet negative relationship for the write-code problem when solved first. There was also a significant positive relationship between problem-solving efficiency and cognitive load for problem five when solved as a Parsons problem first. Yet, an insignificant and negative relationship for the write-code problem (five) when solved second. Did we have achieve a desirable difficulty in the design of problem two and five as Parsons problems

Table 5.12: Correlations between Self-Efficacy Scores and Task Completion Times

		Parsons Problem		Write-code Problem	
Problem	*n*	ρ	*p*-value	ρ	*p*-value
1 has22 (H)	81	-.26	0.021*	-.02	0.873
2 countInRange (M)	84	-.23	0.032*	-.21	0.053
3 diffMaxMin (E)	83	.09	0.440	-0.21	0.052
4 dictTotal (M)	87	-0.15	0.160	-0.28	0.009**
5 dictNames (H)	71	-.14	0.251	0.19	0.111

Notes: * p < .05, ** p < .01, *** p < .001, ρ = Spearman rank-order correlation coefficient when presented to students in a certian order? The element interactivity of problem five was the second highest, problem two was first; problem five had seven correct blocks and two distractor blocks and the topic covered dictionaries. Researchers posit that a combination of varied conditions of practice and worked examples results in effective learning [57]. Code completion effects, which are most related to Parsons problems, are a part of cognitive load theory effects that include worked examples [373]. Thus, we suspect that the intra- and inter-adaptation processes varied the level of difficulty for each individual who solved this problem and that in combination with being a code completion, this led to an optimal zone of development for learning and desirable difficulties. Students may experience less cognitive load and be more efficient at solving Parsons problems with uncommon solutions after solving equivalent write-code problems. Future research should expand on the notion of common/stereotypical to account for different stages of learning. Novices might easily acquire common solutions based on clusters of their peers' write-code solutions but these solutions may not be the expert solutions we are working towards teaching them.

There were a few significant relationships between self-efficacy and problem-solving efficiency for both Parsons problems and write-code problems. Higher self-efficacy scores were correlated with being more efficient at solving problem one (common solution) and problem two (uncommon solution) as Parsons problems and problem four as a write-code problem. This implies that we can use self-reported self-efficacy scores and problem-solving efficiency to improve how the system adapts and scaffolds student learning. Researchers suggest "collecting and combining data from multiple sources can help provide insights that have implications for areas such as learning, instruction, retention, and curriculum design" [328, p. 2]. None of the problems showed a significant relationship between self-efficacy and problem-solving efficiency across problem type even though they were isomorphic. Future research should explore using other measures to capture real time self-efficacy beliefs such as microanalysis and trace data [81].

5.3.4.4 *RQ4: Do students find the modified intra-problem adaptation process understandable and useful?*

We added several questions to an end of course student survey in the winter semester of 2020 and again in 2021 to determine student reactions to the intra-problem adaptation process that was initiated when the student clicked the "Help Me" button on a Parsons problem. In 2020 the adaptation process first removed any distractors from the solution, then provided the indentation if it was needed by adding spaces to the left of each line, and then combined two blocks into one. It selected the blocks to combine based on the number of lines in each block. Students reported that they were confused when the system provided the indentation and did not find it useful when the system combined blocks that were already adjacent. Due to this feedback we modified the intra-problem adaptation process to no longer provide indentation and to combine blocks that were the furthest apart. In the winter of 2020, 17% of the students who answered the end of course survey agreed or strongly agreed that, *"I did not understand what happened after I clicked the 'Help Me' button on the mixed-up code problems"*. In the winter of 2021 we asked the question in a slightly different way, *"I understood what happened when I clicked on the "Help Me" button on a mixed-up code problem"* and this time 7% disagreed or strongly disagreed. This appears to be an improvement, however the wording was different and the response rate was much lower for winter 2021. In the winter of 2021 only 75 of 143 students responded for a response rate of 52.4% whereas 107 of 138 responded in 2020 for a response rate of 77.5%. The difference in response rate is likely due to the course being offered fully remote in the winter of 2021. However, the percentage of students who found the "Help Me" button useful increased from the winter of 2020 to the winter of 2021. In the winter of 2020 79% of the students agreed or strongly agreed with *"I found the "Help Me" button on the mixed up code problems useful"*, while in the winter of 2021, 85.9% agreed or strongly agreed that *"I found the 'Help Me' button on the mixed-up code problems helpful"*. This provides evidence that the changes improved the students' perception of the usefulness of adaptive Parsons problems.

5.4 Think-Aloud Observations

5.4.1 Protocol

We conducted each session on-line via Zoom and began by asking participants for consent and a pseudonym. Participants received a $25.00 incentive for completing the study. First, we asked participants to fill out the prior programming experience survey for context (see Appendix D) [163, 345]. Then, we read them a description of what a think-aloud study is, randomly assigned them to either version A or B of the problem set, and asked them to verbalize their thoughts as they worked through each problem [393]. Three participants did version A and three did version B. The average session lasted forty-two minutes and thirty-seven seconds. Following the session, we conducted a semi-structured interview to understand students satisfaction with the system's learner-initiated help-seeking button. In particular, to understand their reactions to the modified adaptation process.

5.4.2 Analysis

The qualitative analysis was performed using NVivo. Each of the think-aloud observations was transcribed by Rev. We developed a codebook using a hybrid approach [326] of deductive (a priori) and inductive (new) codes listed in Table 5.4.2 [326]. We derived and refined our codes based on previous research [223, 297]. One of the researchers trained with a doctoral student to independently code 50% of the transcripts and identify examples. We calculated Cohen's kappa for inter-rater reliability; it was between 0.70 and 1.00. The remaining transcripts were coded single-handedly. In this paper, we chose to report on the students who solve problem two as a Parsons problem first only.

Table 5.13: Codebook One

Code	Freq.	Definition	Example
Help-seeking	14	Using search engines to get help with a problem or clicking on the "Help Me" button.	*"Okay, I got another error that I don't really understand. Let me Google this."*
Misconceptions:			
Conceptual knowledge	33	Knowledge of specific facts about a programming language and rules for its use.	*"I don't know what the last error—list index out of range—means."*
Strategic knowledge	10	The ability to design, code, and test a program to solve a novel problem. Knowledge of syntactic facts related to a particular language. Ability to apply rules of syntax when programming.	Failure to correctly initialize a variable or merge blocks of code that should be applied together.
Syntactic knowledge	17	Mismatched parentheses, brackets, or quotation marks; irresolvable symbols, missing semicolons, and using illegal start of expressions.	*"Maybe it's a comma, I don't know to be honest."*
Problem-Solving Processes:			
Reinterpret problem	28	Questioning details of the problem prompt or problem requirements.	*"Where is it counting range start and end indices inclusive?"*
Analogous problem search	2	Identifying similarities between the current problem and other problems or solutions.	*"I'm going to go with the key strategy that I learned before."*

Continued on next page

Table 5.13 – *Continued from previous page*

Code	Freq.	Definition	Example
Adapt solution	1	Identifying how a current or prior solution can help solve a current or past problem.	*"It looks like this is the syntax I probably should have used for [the previous problem]."*
Evaluate solution	8	Judging the correctness of code.	*"No. First, I have to iterate through [the list]."*
Self-Regulation Processes:			
Planning	32	Expressing intent to perform some task, or description of a task participants is doing.	*"First, I'm going to define the function."*
Process Monitoring	5	Declaring that a programming sub-goal is complete.	*"I'm going to slice the list into another list....There we go."*
Comprehension monitoring	36	Reflection about the understanding of code or problem prompts.	*"There's no value called index yet. Okay, I have to go inside of loop."*
Management of cognition	2	Decisions about how mental resources are allocated—when to leave a problem for later or stop trying to solve it.	*"I just don't know exactly, so I'm just going to leave it as it is."*
Reflection on cognition	31	Judgments about mental processes, mistakes, assumptions, or biases.	*"I'm pretty bad at this. I mean, assessing how much effort I actually put in."*
Self-explanation	6	An account of why a decision was correct.	*"I figure it's [block 2b] because there's an index."*

5.4.3 Results and Discussion

5.4.3.1 *RQ5: Why did students struggle to solve the Parsons problem with an uncommon solution?*

In this section, we used our qualitative coding scheme to explore how three out of the six *non-computer science* students solved problem two (countInRange) as a Parsons problem (see Figure 5.3). The solution for this problem was not the most common student written solution. We sought to understand why students struggled to solve this problem and if they used the Parsons problem solution to solve the equivalent write-code problem.

5.4.3.2 Solved Parsons then Write-Code Problem

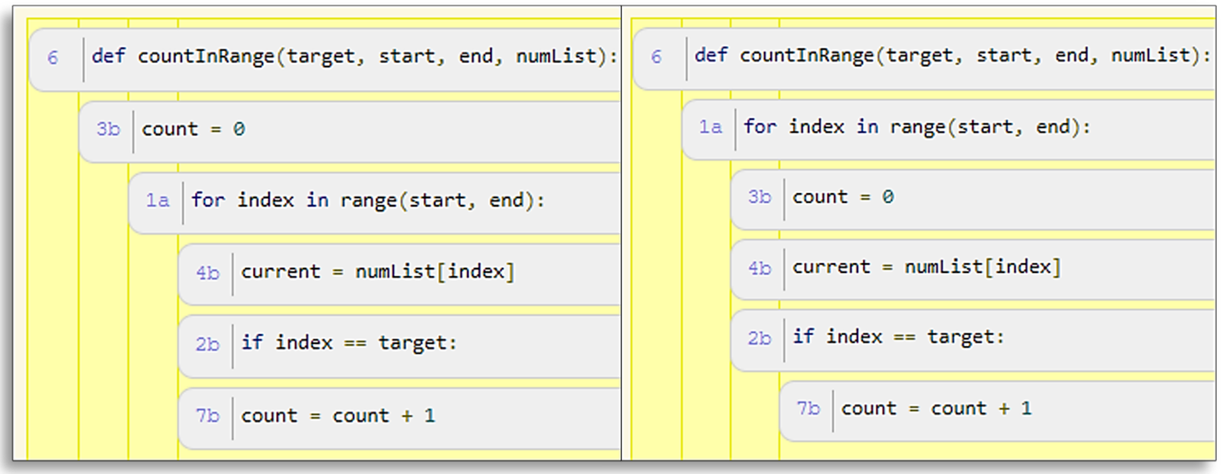
5.4.3.3 Logan

Logan was a 20-year-old who identified as male and chose not to indicate his race. He was a senior theatre performance major who was specializing in acting and he had 5 months of prior programming experience which he gained from a semester taking a non-computer science college course. On the university's math placement test, he scored a 22 out of 25 and his overall GPA was 3.7. Based on his self-efficacy scale ratings, he was grouped into the low average cluster (see Table 5.2). Of the six participants, Logan had the lowest score (2.8) for "persistence, debugging, and problem-solving competences" (Factor 3) on the self-efficacy scale (see IPSES for the complete scale). Students were asked questions such as, "Given the design of a solution and an incorrect program, can you identify the source of the error?" Logan solved problem two in three hundred and twelve seconds, approximately three minutes slower than the median (Table 2).

When he began to solve problem two, Logan had trouble understanding the prompt. He highlighted **`countInRange(1, 2, 4, [1, 2, 1, 1, 1, 1])`** with his mouse and said, "I'm trying to understand this whole countInRange thing. [The function] should return three since there are three ones between index two and four. Where is it counting range start and end indices inclusive? Are they saying these are the indices? I don't really understand why it's three ones since there are four here. Are you able to explain exactly which ones are the indexes and which ones are the list that they're referring to?" Logan tried to reinterpret the problem, experienced a conceptual misconception, and also engaged in help-seeking; he was confused about the parameter values being passed to the function. He then said, "Oh, start and end indices inclusive. Okay. Zero, one, two, three, four. Yeah, so it'd be three [ones]." He realized the list index started with zero and said, "Wow, that's just over-complicating things—in my opinion."

Logan moved on to select blocks 6 and 3b correctly. But when choosing between blocks 1a and 1b (the two for loops), he grabbed the wrong block 1a, **`for index in range(start,`**

end):, indented it incorrectly, and said, "Let's try this." He'd forgotten that to be inclusive of the end he needed to choose the for loop with the default argument **end+1**. This was both a conceptual and strategic misconception. Logan then read over blocks 4a **current = numList[start]** and 4b **current = numList[index]**, and said, "What the heck is that?" When prompted by the researcher to explain what he was thinking, Logan said, "[Block 4b is the right one] because we're starting at the index and it's going to iterate through. Actually, it looks like this is the syntax I probably should have used for [the previous problem], which I might go fix later." He engaged in self-explanation when prompted and realized something about the previous write-code problem (has22). To manage his cognition, he had left problem one (has22) incomplete and moved on to this problem. He planned to adapt the current solution for this Parsons problem two to write-code problem one (has22). Logan then reread blocks 1b and 4b before he chose between blocks 2a and 2b. Without prompting, he engaged in self-explanation and planning. He said, "I figure it's [block 2b] because there's an index. I'm going to make it easy for myself and assign it to current. And **if current == target**, I can do **count = count + 1**. Wait, wait, wait...I can't indent anymore." When Logan realized he could not indent block 7b, he reformatted his solution (see 5.6). This led to a strategic misconception; at first, Logan initialized the count variable correctly, but then he placed block 3b into the for loop underneath block 1a.



```
6   def countInRange(target, start, end, numList):
3b      count = 0
1a      for index in range(start, end):
4b          current = numList[index]
2b          if index == target:
7b          count = count + 1
```

```
6   def countInRange(target, start, end, numList):
1a      for index in range(start, end):
3b          count = 0
4b          current = numList[index]
2b          if index == target:
7b              count = count + 1
```

Figure 5.6: *Logan's strategic misconception of problem two.*

Next, Logan reread blocks 1a and 1b. He then said, in reference to block 1a, "I'm not even sure about this range. I don't know if that's right." This confirmed that he did experience a conceptual misconception when choosing between the correct for loop block 1b and the distractor block 1a. Logan then chose block 5 **return** and clicked "Check". He received an error message that said, "Highlighted blocks in your program are wrong or are in the wrong order. This can be fixed by moving, removing, or replacing highlighted blocks." Block 1a was highlighted. He said, "It's probably this one then"—in reference to block 1b. Then he clicked "Check" again and received the same error for block 1a; this error was a conceptual and strategic misconception in that he did not understand that the order of statements would result in assigning the count variable to zero each time. His solution would not keep track of how many times the target appeared between the start and end because count would be reset to zero after looping through the list.

Finally, Logan moved block 1b to the correct position before checking his solution again. This time he experienced a syntactic misconception; Logan had placed the **`return count`** (block 5) in the correct order but did not know how it was supposed to be indented. A popup window appeared that said, "Click on the Help Me button if you want to make the problem easier." He clicked the "OK" button but did not click the "Help Me" button; he chose not to seek help. Block 5 **`return count`** was then highlighted to suggest that it be indented. Logan indented block 5, clicked "Check," and rated investing "neither low nor high effort" in solving the problem.

Logan engaged in help-seeking as soon as he had trouble interpreting the problem prompt but asked the researcher instead of searching the web and declined help from the system toward the end; he also engaged in trial-and-error at the start without much planning or comprehension monitoring. This led Logan to experience several misconceptions; he was misled by the distractor block. This block was meant to teach students that the loop must change if you want the end of the range to be inclusive. Prior research confirms that novice students who struggle with computer programming go to tutors instead of trying to understand problems on their own by searching the web as more advanced students do [218]. And, furthermore, students with significantly low self-efficacy have misconceptions about computer programming functions [179]. Students like Logan may benefit from guidance in the form of subgoals to help them plan better and from an explanation of why distractor blocks are incorrect.

When asked about his preferences for adaptive Parsons Problems vs. write-code problems? Logan said, "I prefer adaptive Parsons problems. They give you what you need syntactically....but when you get to writing portions, it's that much harder if you're constantly given these jumbled up problems....I wish there was a way that not only did you have to [drag-and-drop]...but that you also had to type it. I think the energy to type it as you put it into the box may help in the long run for [write code problems]. It's like a transition to the [write-code problems]."

When asked about the adaptation process? Logan said, "Yeah, I thought [the help-seeking features] were helpful...sometimes it's annoying....I'm not a huge fan of combining blocks...I would love to see the contrast....I wish there was a load history for [adaptive Parsons problems] too." The write code problems have a "load history" button which loads all versions of the code that were submitted to be executed and the student can use a slider to review these versions.

5.4.3.4 Radhamani

Radhamani was a 19-year-old Asian who identified as female. She was a junior business administration major who had three months of prior programming experience in non-computer science college courses. She scored a 25 on the university's math placement test; her GPA was 3.8. Like Logan, Radhamani was sorted into the low average cluster based on her self-reported self-efficacy scores; her score was 3.2 for "persistence, debugging, and problem-solving competences". It took her two hundred seconds to solve problem two, one minute and fifteen seconds slower than the median (Table 2).

Radhamani engaged in planning right after reading the prompt. She said, "First, I'm going to define the function." Then she selected blocks 6 and engaged in self-explanation and comprehension monitoring. Radhamani said, "This [block 6] is the only block with the definition, and I see the first thing you feed in is the target, then the start index, and the end index, and then the list itself." Then, she initialized the count variable—block 3b—and engaged in more planning and comprehension monitoring which prevented her from experiencing a strategic misconception but not a conceptual misconception. She said, "I'm just looking through the options. Okay, so I want to set current to equal the item in the list [block 4b]. Oh wait. There's no value called index yet. Okay, that [block 4b] would have to go inside a loop, so for index in range start end...I'm debating between these two options [blocks 1a and 1b] right now. I think it's start end, so this is saying, for any index in this range, I think the end is inclusive but if that's wrong, I'll switch it out." Radhamani mistakenly selected block 1a just as Logan did. She understood that the end must be inclusive, but did not understand the range method ends with the end minus one index (i.e., it is not inclusive).

Finally, Radhamani engaged in some more planning and selected block 4b, then block 2a—which she initially indented incorrectly but caught it on her own—block 7b, and block 5. She then checked her solution, received an error and replaced block 1a with the correct for loop block 1b. When prompted to rate how much mental effort she invested in solving the problem, she reflected on her cognition and said, "This took me a lot less effort than that first [problem] just cause I didn't get stuck on something." She rated investing "low mental effort" in solving it.

Radhamani engaged in planning, comprehension monitoring, and self-explanation from the start. These processes helped her avoid some pitfalls. She didn't get as stuck although she was still distracted by the same for loop [block 1a] as Logan. Prior research on self-regulated learning in programming shows there is a significant positive correlation between students' use of metacognitive and resource management strategies and programming performance [29]. The more students plan, the better they do. Furthermore, researchers posit there is a need for "consistent, disciplined self-regulation during problem-solving" such as asking students to self-report cognitive load [223, p. 90].

When asked about her preferences for adaptive Parsons Problems vs. write-code problems? Radhamani said, “I think the drag and drop ones (Parsons problems) are easier. It’s more helpful as a starting point. It’s helps with not having to remember how to actually define the stuff, but I think actually writing it out helps me learn more because it forces me to Google it and then I actually learn the syntax myself. I prefer typing it out.”

She valued the solution to Parsons problem two and used it to solve problem one. Radhamani said, “This [problem two’s `for num in range(len(nums) - 1):`] kind of taught me that this value is excluded and this one is included. And that helped me do what I just did here with problem one (has22)....excluding the negative one.

When asked about the adaptation process? Radhamani said, “I guess the distractors are more to check your understanding, but first you have to actually understand it...so removing those blocks is helpful....If I’m really struggling, I might choose to have that distractors removed, but if I feel like I’m on the verge of solving it, I might prefer to keep them in there and maybe just get a more general hint.”

5.4.3.5 Izaan

Izaan was a 19-year-old Asian sophomore who identified as male. He hadn’t declared his major yet. Izaan had one year of prior programming experience that he gained through two semesters of college courses in computer science. He scored an 18 on the university’s math placement test and his GPA was 4.0. Izaan had the highest self-reported self-efficacy score (4.4) for belief in ones “persistence, debugging, and problem-solving competences” (Factor 3). His other scores were: 5.833 (Factor 1), 6 (Factor 2), and 4.833 (Factor 4). This put Izaan in the average high cluster. He completed the problem in three minutes and forty seconds, one minute and thirty-five seconds slower than the median (see Table 5.4).

When Izaan began to solve problem two, he started monitoring his comprehension immediately. He said, “The first parameter is the number that’s the target value. The second one is the first index to look at and four is the last index.” Then he engaged in planning and self-explanation. Izaan said, “First, what I’m going to do is define [the function] and there’s only one [block] to define it.” He correctly chose block 6. Next, he engaged in reinterpreting the problem and comprehension monitoring while choosing between blocks 3a and 3b; he questioned, “so you probably have to initialize count to zero?...returns a count of the number of times that the target value appears. Let’s try initializing that first”; he chose block 3b correctly.

Izaan then chose to assign the variable **`current = numList [start]`**—the distractor block 4a. He caught this conceptual and strategic misconception because he stopped to evaluate his solution while planning his next move. He said, “Oh shoot. No. First, I have to iterate through [the list].” He removed block 4a, and unlike Logan and Radhamani, Izaan chose the correct for loop (block 1b) while engaging in self-explanation. He said, "For index in range from start to end plus one...end plus one, so you have to add one for the index." Yet, Izaan still chose block 4a next, which showed he was still experiencing a conceptual misconception. This block incorrectly passed the start index parameter from the range method to numList. Izaan continued to engage in planning and comprehension monitoring and caught this conceptual misconception. He said, "Current is equal to numList at start...**`if current == target`** (block 2a). **`if index == target`** (block 2b). Current is equal to numList...For index in range...Oh shoot so that should be **`current == numList[index]`**."

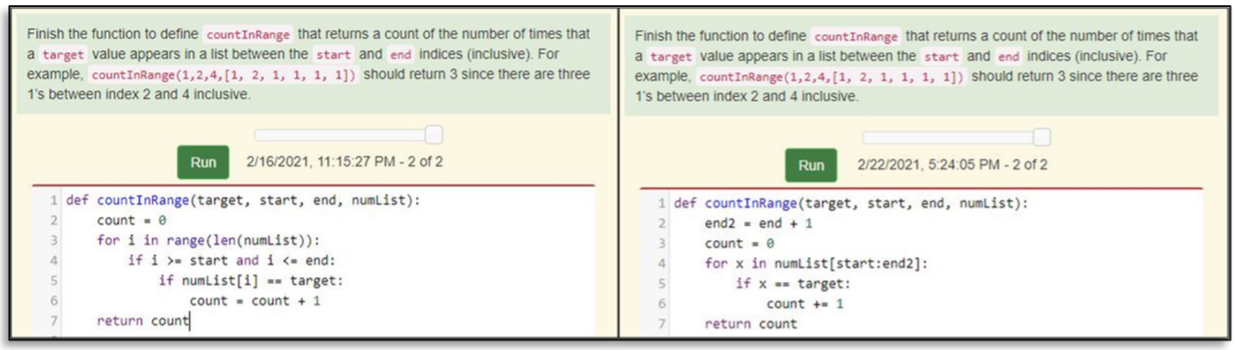
He chose block 2a and indented it correctly. Then he engaged in more planning and experienced a misconception about how the interface worked regarding distractor blocks. Izaan said "And then if the current is equal to target [block 2a], indent, then we're going to do count. **`count++`** (block 7a) or **`count = count + 1`** (block 7b), should be the same thing—I'm just more comfortable with **`count = count + 1`**. And then at the end, we're going to want to **`return count`** [block 5], which would go outside of [the loop]." He did not realize that the 'or' connecting blocks 7a and 7b meant that one of them was a distractor block; he thought they were both correct. Finally, Izaan rated investing "neither low nor high mental effort" in solving the problem when reflecting on cognition.

When asked about his preferences for adaptive Parsons Problems vs. write-code problems? Izaan said, "Obviously, from a lazy point of view, I always prefer [adaptive Parsons problems], but in terms of when I'm actually trying to learn a concept, I found that those don't actually help you very much, because it's like a process of elimination at the end of the day and the ones where you actually have to write the code [are] a lot more challenging and it makes you think a lot harder. Usually, that's the way I try to...If I'm trying to learn something, I'll usually do those problems."

When asked about the adaptation process? Izaan said, "Usually, it takes a couple blocks out if you have the wrong blocks inside of it or it'll combine blocks, which is really helpful....I think if there weren't distractors in the mix-up code problems they wouldn't be very helpful at all because it's more a test of how many extra things you can put [in order]."

Radhamani and Logan both completed version A of the problem set for extra credit after their think-aloud observation; Izaan didn't. In that version, problem two (countInRange) was presented to them as a write-code problem. They both solved it using a different solution than the Parsons problem solution (see Figure 5.7). Each of them had trouble understanding which of the two for loop blocks was inclusive. This could explain why the students who wrote the code first were more efficient at solving the Parsons problem than the students who solved the Parsons problem first. Distractors can slow the problem-solving process [145].

Since some students, like Radhamani and Izzan, find more value in writing code than in solving a Parsons problem, we have added the ability to switch from a Parsons problem to an equivalent



Finish the function to define `countInRange` that returns a count of the number of times that a `target` value appears in a list between the `start` and `end` indices (inclusive). For example, `countInRange(1,2,4,[1, 2, 1, 1, 1, 1])` should return 3 since there are three 1's between index 2 and 4 inclusive.

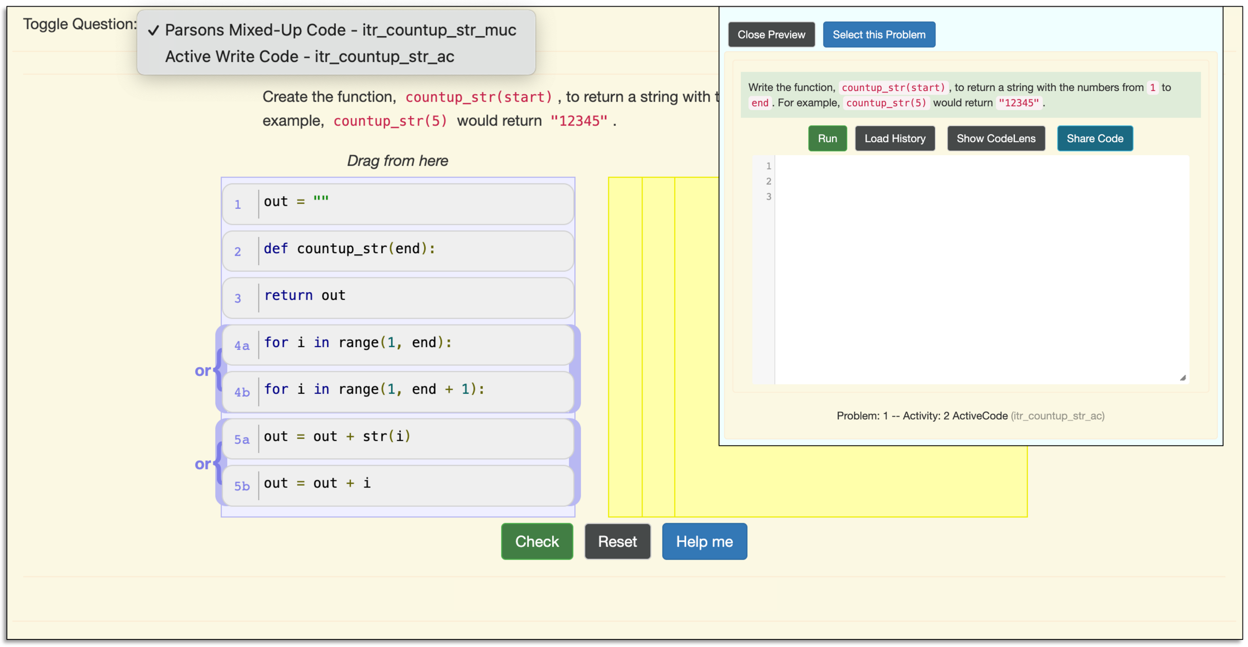
```python
def countInRange(target, start, end, numList):
    count = 0
    for i in range(len(numList)):
        if i >= start and i <= end:
            if numList[i] == target:
                count = count + 1
    return count
```

Finish the function to define `countInRange` that returns a count of the number of times that a `target` value appears in a list between the `start` and `end` indices (inclusive). For example, `countInRange(1,2,4,[1, 2, 1, 1, 1, 1])` should return 3 since there are three 1's between index 2 and 4 inclusive.

```python
def countInRange(target, start, end, numList):
    end2 = end + 1
    count = 0
    for x in numList[start:end2]:
        if x == target:
            count += 1
    return count
```

Figure 5.7: *Logan's (left) and Radhamani's (right) Alternative Solutions to Problem 2.*

write-code problem with unit tests (see Figure 7.2). We are currently testing the effect of this type of problem. Students can get credit for solving either type of problem. This will allow us to challenge the high end students while still supporting struggling students.



Toggle Question:
✓ Parsons Mixed-Up Code - itr_countup_str_muc
Active Write Code - itr_countup_str_ac

Create the function, `countup_str(start)`, to return a string with t
example, `countup_str(5)` would return `"12345"`.

```
1   out = ""
2   def countup_str(end):
3   return out
4a  for i in range(1, end):
4b  for i in range(1, end + 1):
5a  out = out + str(i)
5b  out = out + i
```

Write the function, `countup_str(start)`, to return a string with the numbers from `1` to `end`. For example, `countup_str(5)` would return `"12345"`.

Problem: 1 -- Activity: 2 ActiveCode (itr_countup_str_ac)

Figure 5.8: *Runestone's Toggle Question Feature*

Students like Logan value Parsons problems but want opportunities to write some of the code and could benefit from Parsons problems designed to support planning [121]. One of the authors is developing a computer programming practice environment, called Codespec, that offers multiple means of engagement to optimize choice between problem types (see Figure 7.1). Its problem space area offers learners the option to switch between solving a problem as either a pseudocode problem (also described as subgoals or programming plans) [254, 72], Parsons problem [283, 98], Faded Parsons problem [405], fix code problem [102], or write code problem. Preliminary results show the average System Usability Scale (SUS) score was above average (*M* = 77.14 out of 100, *SD* = 6.03) [15]. We plan to explore questions like when selected response (i.e., solving a Parsons problem with only one solution or a selected problem type) is and is not 'cognitively equivalent' to open response (i.e., solving a problem with many solutions or choosing between problem types).

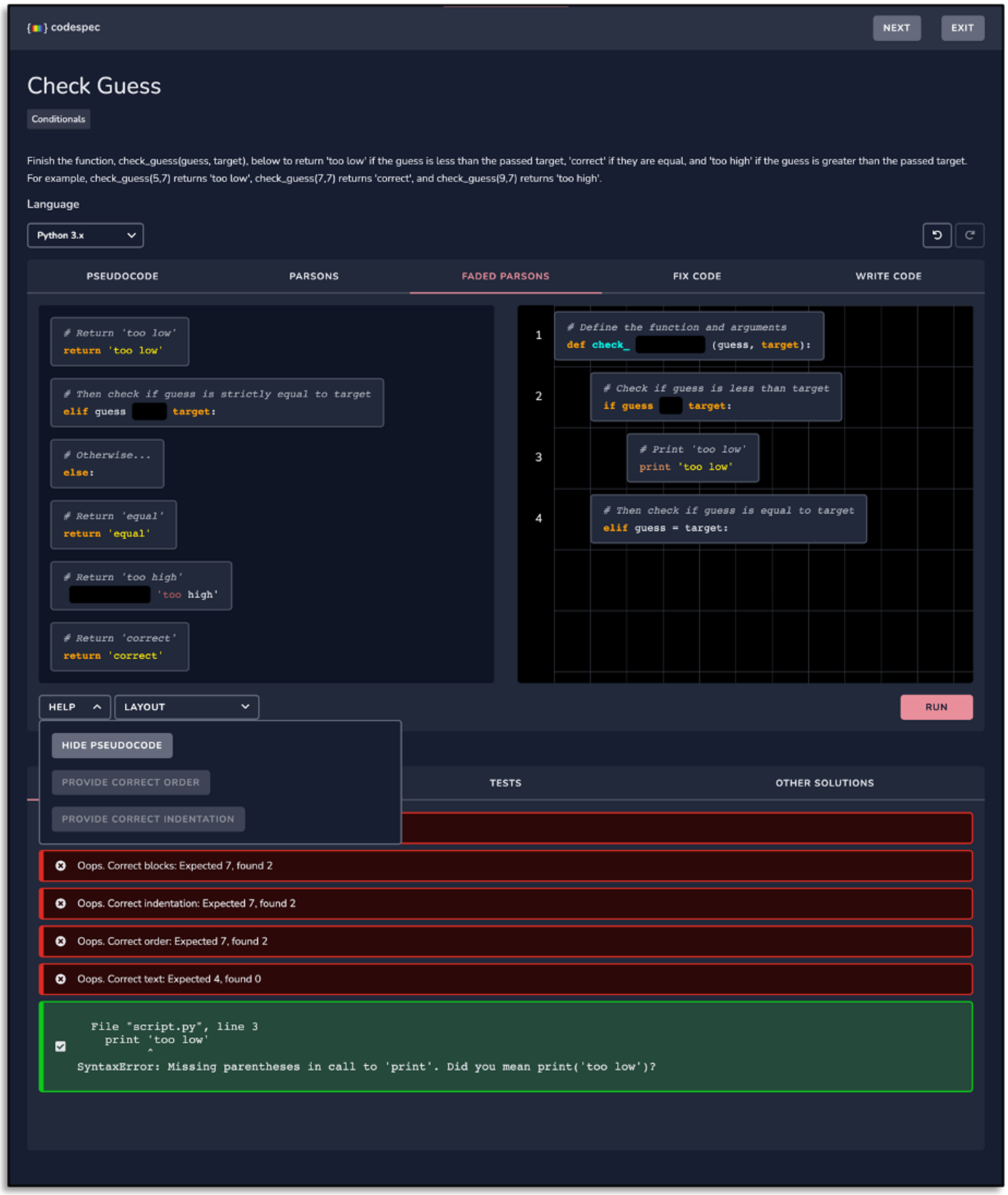
To help students who are struggling while writing code from scratch we are also testing allowing them to view or solve a Parsons problem as a type of hint. They would still be required to solve the write code problem. They can solve the Parsons problem, but they can not just copy and paste the Parsons solution to the write code problem. They would at least have to retype the Parsons problem solution in the write code problem. This is in line with the recommendation from the student identified as Logan.

The think-aloud observations with semi-structured interviews also raised some possible areas to explore. We could investigate adding a slider to the Parsons problems to allow the learner to review how the solution changed over time. We have already added the ability to explain why a distractor block is incorrect, but have not tested this change. We could explore forcing the learner to do more before providing additional help.

5.5 Limitations

This study was conducted in one course at a research intensive university in North America using Python. The study should be replicated in other contexts and languages. This study provides evidence that Parsons problems are not more efficient than writing the equivalent code when the Parsons problem solution is uncommon, but are when the solution matches the most common student written solution. Yet when students solved the Parsons problem with an uncommon solution first, it led to the largest learning gains. One of the most persistent ways to capture learning gains is by using pre-post test, but its computation is heavily debated [314]. One of the limitations to measuring learning gains in this uncontrolled environment is that students could have just copied the solution from the Parsons problem. Future research should explore other ways to compute learning gain scores [314] in controlled settings. Furthermore, we only tested one problem with both an uncommon solution and the most common solution. Additional studies should be done with both common and uncommon solutions. Researchers should also explore fine-grained time-on-task metrics [214] and other ways to compute efficiency [159], measure cognitive load [92], and capture self-efficacy beliefs [81].

While a lower percentage of students reported not understanding the Parsons problem adaptation process and a higher percentage found the adaptation useful in the most recent end of course student survey, the response rate was much lower in the winter of 2021 than in the winter of 2020.



Check Guess

Finish the function, check_guess(guess, target), below to return 'too low' if the guess is less than the passed target, 'correct' if they are equal, and 'too high' if the guess is greater than the passed target. For example, check_guess(5,7) returns 'too low', check_guess(7,7) returns 'correct', and check_guess(9,7) returns 'too high'.

Language: Python 3.x

```
# Return 'too low'
return 'too low'
```

```
# Then check if guess is strictly equal to target
elif guess  target:
```

```
# Otherwise...
else:
```

```
# Return 'equal'
return 'equal'
```

```
# Return 'too high'
 'too high'
```

```
# Return 'correct'
return 'correct'
```

```
# Define the function and arguments
def check_ (guess, target):
    # Check if guess is less than target
    if guess  target:
        # Print 'too low'
        print 'too low'
    # Then check if guess is equal to target
    elif guess = target:
```

```
Oops. Correct blocks: Expected 7, found 2
Oops. Correct indentation: Expected 7, found 2
Oops. Correct order: Expected 7, found 2
Oops. Correct text: Expected 4, found 0
```

```
  File "script.py", line 3
    print 'too low'
    ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print('too low')?
```

Figure 5.9: *Codespec's Problem Space Area on the Faded Parsons Problem with Pseudocode*

This was likely due to the pandemic. The survey should be run again in future semesters.

5.6 Conclusion

Parsons problems were created to provide engaging practice for novice programmers that was easier than writing the equivalent code from scratch [283]. Our think-aloud observations show that students do perceive them as easier but some preferred write-code problems for learning. Parsons problems have been used to help students learn syntax, common algorithms, common errors, and new approaches to solving problems [89]. They can have significantly lower cognitive load than writing the equivalent code, but not always. Prior research has shown that most students find adaptive Parsons problems helpful for learning though some students would rather write the equivalent code. This research has provided evidence that Parsons problems with uncommon solutions are not significantly more efficient to solve than writing the equivalent code, but led to larger learning gains. It also provided evidence that Parsons problems with the most common student written solution are significantly more efficient to solve than writing the equivalent code. This implies that the best way to generate Parsons problems is to cluster student written code and use the most common student solution. This is one way to assess students current competencies efficiently, without frustrating them and possibly boosting their self-efficacy beliefs, before we present them with Parsons problem solutions that are uncommon which might naturally lead to learning gains. "We learn 'at the edges' of what we already know by adding to existing knowledge" [311, p. 351]. It is possible that the most common student solution changes over time; problem two was significantly efficient to solve in a prior study, but was not in this study. This implies that we may need to account for context and other types of data to dynamically generate Parsons problems with respect to desirable difficulties. We plan to add the ability to dynamically generate Parsons problems from student written code and use Parsons problems to provide more efficient practice than writing the equivalent code as well as to scaffold students who struggle while writing code. We also plan to explore multiple means of engagement with programming practice that include a range of researched based problem types.