1 SUPPLEMENTAL MATERIAL

1.1 Problem Set with Instructions and Solutions

1.1.1 Problem One. Finish the function **has22** below to return **True** if there are at least two items in the list **num**s 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.

The adaptive Parsons problem solution presented to students and the most common student written solution:

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

1.1.2 Problem Two. 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.

The adaptive Parsons problem solution presented to students:

```
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
```

Most common student written solution for participants who solved this problem as a write-code problem first:

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

OverCode clusters of solutions for students who solved this problem as an adaptive Parsons problem first (see Table ??):

Cluster 1 (n = 7):

```
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
```

Cluster 2 (n = 6):

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

Cluster 3 (n = 6):

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

Cluster 4 (n = 5):

```
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
```

Cluster 5 (n = 2):

```
def countInRange(target, start, end, numList):
    return numList[start:end+1].count(target)*
```

Cluster 6 (n = 2):

```
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 += 1
    return (count)
```

1.1.3 Problem Three. Finish the function diffMaxMin to return the difference between the largest and smallest value in the passed list of numbers (nums). For example, diffMaxMin([1,2,3]) should return 2 since the difference between 3 and 1 is 2.

The adaptive Parsons problem solution presented to students and the most common student written solution:

```
def diffMaxMin(nums):
    large = max(nums)
    small = min(nums)
    return large - small
```

1.1.4 Problem Four. Finish the function `total_values` that takes a dictionary (dict) and returns the total of the values in the dictionary. For example, `total_values`('red': 3, 'blue': 2, 'green': 20) would return 25.

The adaptive Parsons problem solution presented to students and the most common student written solution:

```
def total_values(dict):
    total = 0
    for key in dict:
        total += dict[key]
    return total
```

1.1.5 Problem Five. Finish 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'. For example, ['first': 'Ann', 'last': 'Brown', 'first': 'Darius'] should return ['Ann Brown', 'Darius Unknown'].

The adaptive Parsons problem solution presented to students and the most common student written solution:

```
def get_names(list_of_dict):
    name_list = []
    for p_dict in list_of_dict:
        first = p_dict.get('first', 'Unknown')
        last = p_dict.get('last', 'Unknown')
        name = first + "␣" + last
        name_list.append(name)
    return name_list
```

1.2 Introductory Programming Self-Efficacy Scale (IPSES)

We used the Introductory Programming Self-Efficacy Scale (IPSES) [1] [**?**] 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 [**?**] and report both Cronbach's α and McDonald's ω for the both administrations of the scale given methodological disputes [see **? ?**]. 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 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 [**?**] with the exception that Factor 4 was lower than 0.95 as recommended [**?**].

Table 1: 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.

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

1.3 Paas Scale

In solving the proceeding problem, I invested (choose one of the following ratings):

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

1.4 Prior Programming Experience Survey

Overall Experience

1. How much experience (i.e., months, years) have you had programming?

High School Experience

2. Did you learn to program as part of a high school class (either formal or informal)?
3. If yes, how many semesters did you take courses involving programming?
4. Was this programming experience part of another course (e.g. Math, Business, Science...)?

College Experience

5. Did you learn to program as part of a college course?
6. If yes, how many semesters did you take courses involving programming?
7. Was this programming experience part of another course (e.g. Math, Business, Science...)?

Work Experience

8. Did you program for work or an internship?
9. If yes, was your work programming experience full time, part time, or less than part time?
10. Roughly how long did you have this work and/or internship programming experience?

1.5 Average Task Completion Times for Parsons → Write

Table 2: Task Completion Times for Parsons → 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 ≡ indicates that students who solved the adaptive Parsons problem first used the same solution to solve the equivalent write-code problem.

1.6 Average Task Completion Times for Write → Parsons

Table 3: Average Task Completion Times for Write → 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 ≡ indicates that students who solved the adaptive Parsons problem first used the same solution to solve the equivalent write-code problem.

1.7 Codebook

Table 4: 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."*
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."*

1.8 Think-Aloud Observations

1.8.1 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 1). 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 block 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 1). 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.

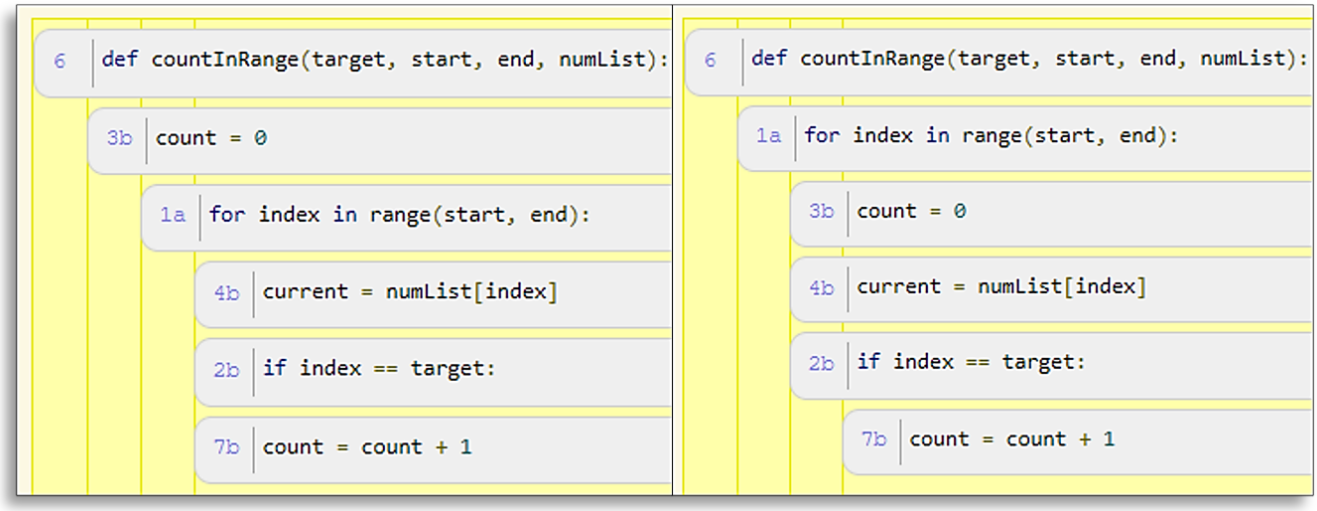

Figure 1: Logan's Strategic Misconception of Problem 2.

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 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 [**?**]. And, furthermore, students with significantly low self-efficacy have misconceptions about computer programming functions [**?**]. 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."

1.8.2 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 [**?**]. 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 [**?** , p. 90].

When asked about his 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 help 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."

1.8.3 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 **??**).

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 2). 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 [**?**].

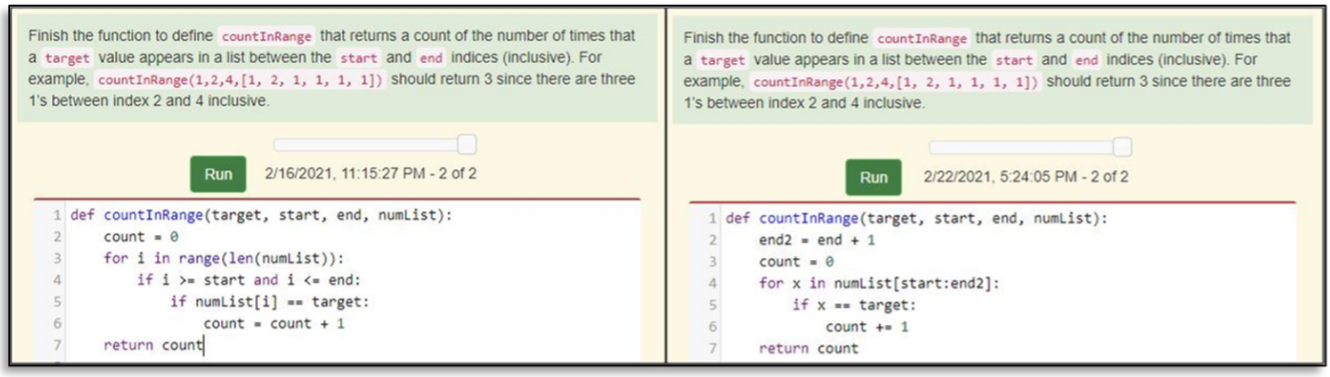

Figure 2: Logan's (left) and Radhamani's (right) Alternative Solutions to Problem 2.