Novice Programmers Difficulties when using Pseudocode as an Intermediary Step in Problem Decomposition*

Elvis Kahoro[1], Isaiah Bresnihan[2]
[1]Pomona College
Claremont, CA 91711
{elvis.kahoro}@pomona.edu
[2]Harvey Mudd College
Claremont, CA 91711
{ifujiibresnihan}@g.hmc.edu

Abstract

Pseudocode can improve ultimate answer quality and help the problem decomposition process. We investigate pseudocode misconceptions that could be detrimental to novice computer science students ultimate answer quality. We conducted an exploratory study that involved one-one-one interviews with eight undergraduate students. Each student had taken at least a semester of computer science but no more than three semesters. The students were given five questions that test problem decomposition. This analysis focuses on how they transformed a previous solution to account for the introduction of new paramaters and the role that pseudocode plays. We hypothesize that students' perceptions of pseudocode can lead to implicitly overlooking and or explicitly ignoring problem structure.

*Copyright ©2018 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.

1 INTRODUCTION AND RESEARCH

Novice computer science (CS) students have difficulty developing algorithmic solutions while adhering to the syntactic constraints of the target programming language.[4] Pseudocode allows students to write syntax-free solutions releasing cognitive load spent on syntax recall and code compilability; this puts emphasis on pattern recognition and problem decomposition.[6] Copus claim that the objectives of pseudocode production are (1) to capture all the elements of the solution to a computing problem in a language somewhere between natural language and code, and (2) to do so at a level of abstraction that is neither too low, so that it is essentially computer code, nor too high, so that it is essentially a restatement of the computing problem. [1] Pseudocode can thus play an important intermediate step in the decomposition of a problem.

Copus finds a significant positive correlation between pseudocode quality and ultimate answer quality, with $r(14) = .590, p < .01$ (one tailed).[1] Teaching students to use pseudocode as a tool to identify design patterns and can be fruitful. Lahtinen finds that the issues relating to understanding programming structures, learning the programming language syntax, understanding how to design a program to solve a certain task, and dividing functionality into procedures, functions and/or classes all have a strong positive correlation with each other (0.534 ¡ r ¡ 0.637, p = 0.01).[5] The practice of producing quality pseudocode coupled with pattern-oriented instruction can help students (1) recognize similarities between novel problems and previously solved problems; and (2) observe the "essential picture" by identifying its components and the relationships between them.[6] In Fidge's study which tasked students with "explaining in plain English" how they would find the minimum and maximum values in a bag of marbles, some students wrote Python code first and then converted their programming solution into English.[6] This approach reverses the role that pseudocode is supposed to play in the decomposition process and motivated us to investigate the kinds of unexpected outcomes that present themselves when students choose to write pseudocode over Python and Java. We intentionally left pseudocode undefined and up to the student's interpretation to gain insight on how assumptions of what constitutes pseudocode influenced their interaction with the problem.

2 METHODS

We interviewed eight computer science (CS) undergraduates who have taken the introduction to computer science course (CS1), who were familiar with the Python or Java programming language, and were not passed the third computer science course (CS3).

2.1 Data Collection

All interviews were one-on-one and lasted approximately an hour. Interviews were filmed with student consent and any scratch work from the interview was collected. Each student was given a packet of five problems to answer during the interview. The interviewer emphasized that the end goal was not the "correct" solution but insight into problem decomposition. The interviewer asked that the student vocalize their thought process and we answered clarifying questions about the problem while aiming to maintain a neutral position, abstaining from leading the student towards any particular solution. The interviewer often asked for an explanation of the student's solution.

2.2 Data Analysis

We used a grounded approach[2] to develop hypotheses about the elements of problem decomposition that students might miss or ignore and the aspects of writing pseudocode that promoted this. For each interview, we created a content log[3], a document that time-stamps significant events throughout an interview. These content logs helped us determine the content of our analysis.

We noted multiple episodes from four of our interviewees in which they asked for clarification on the definition of pseudocode, remarked that producing pseudocode was difficult, and in which they attempted to use Java or Python, but "resorted" to pseudocode. We conducted further analysis of these episodes using the content logs, video recordings, and the students' written work. This motivated us to investigate the potential gaps in ultimate answer quality produced by low quality pseudocode and the freedoms that the student perceives pseudocode affords them.

2.3 Question Description and Solution

Rainfall Problem: Write a program that continuously reads in (positive) integers from a user, until the user enters 99. After 99 is read, it prints out the average of the numbers entered previously, NOT counting 99. If the input is negative at any time, ignore it and do not add it to the running sum, then continue to accept input. If a user enters: [3, 6, 7, -3, 8, 100, 99, 35] the program returns 24.8.

This question tests three algorithmic patterns: Exists?, Do All? and Succ-Trav.[1] Exists?: Checks for the existence of an item that satisfies a condition. The student should check for 99 in order to trigger the termination sequence. Do All?: Checks if all items satisfy a condition. The student only includes positive values. Succ-Trav: Traverse successive elements. The student successfully generates a loop so the solution can be generalized to an input of any size.

Listing 1: Rainfall Solution in Python

```
runningSum = 0
count = 0

def rainfall(sumList):
    for num in sumList:
        if num == 99:
           break
        elif num > -1:
            runningSum = runningSum + num
            count = count + 1
    return runningSum / count
```

Rainfall Bonus Problem: We followed up the rainfall problem with a variation that tests the programming pattern Num-to-Digits. Did they reuse most of their previous code? How did they morph their previous solution to account for this new condition? A combination of the latter or did they start entirely from scratch? Our presented solution to this problem consists of creating a helper function that converts the number into an array of digits and then adding an inner for loop to the original solution.

Listing 2: Rainfall Solution in Python

```
runningSum = 0
count = 0

def numToDigits(numToSplit):
    digits = []
    while numToSplit > 0:
        digitsArray.append(numToSplit % 10)
        numToSplit = numToSplit / 10
    return digits

def rainfall(sumList):
    for num in sumList:
        if num == 99:
           break
        elif num > -1:
            digitList = numToDigits(num)
            for digits in digitList:
                runningSum = runningSum + digits
                count = count + 1
    return runningSum / count
```

3 ANALYSIS

In this section we analyze the episodes from the four interviews. The first episode comes from an interview with Catherine, who expressed confusion about what it means to write pseudocode. The second episode comes from an interview with Sylvia, who modified the input of the problem to not include numbers larger than 100 instead of creating a general solution that works for scope of any input size. The third episode comes from Tasha who morphed her original rainfall solution to vaguely capture the essence of the bonus. The fourth episode comes from Davida who resorted to writing pseudocode because of difficulty recalling language specific syntax. ***Note*: Bold lines numbers are of the interviewer responding.**

3.1 Episode One - Catherine

Catherine answered the first part of Rainfall quickly and correctly in Java but shifted from Java to pseudocode when starting the bonus Rainfall problem.

	Dialogue
1	I don't know how to write pseudocode.
2	I feel like I'm just writing a paragraph on different lines and putting semicolons on them, at the end.
..	*(Student tries to decide whether to take input as string or list of ints)*
4	Is this pseudocode?
5	I'm sorry, this is a clarifying question.
6	**It can be in anything, like java, python or c++.**
7	I feel like it'd be easier to do pseudocode because I don't know exactly how to parse anymore.
8	Is this pseudocode though, or is this just writing in paragraph on different lines?
..	*(Excerpt isn't applicable)*
9	How about that, how's that semicolon.
10	I feel like that just made it pseudocode.
11	Really high level language this one is.
..	*(Reads aloud what she's written so far)*
12	That would be alot easier if I just wrote that in actual code, but you know what we're all gonna do this pseudocode thing.

Catherine had difficulty with writing pseudocode (line 1) and her version of pseudocode leaned towards the natural language side of the spectrum (lines 2, 9, 10, 11). She shifted from Java to pseudocode, despite her success with Java in the previous problem because of difficulty recalling the semantics of parsing input from the user in Java. She remarks that it would be easier to "write actual code" (line 12) but felt as if she has made it far enough along the process that it was not worth switching back to Java and rewriting her solution.

Listing 3: Catherine's Pseudocode Solution

```
a double harderAverage {ArrayList<String> str) {
b create sum variable set equal to zero;
c create counter variable set to zero;
d input from user read as string;
e    for loop that goes through the list at each index {
f        for loop that parses the string of each new letter {
g              sums the nested strings (holding) just one
               character into  ints that are plus equaled
               into sum;
h        counter++ at each digit;
i        }
j   }
k   return sum / counter * 1.0;
l }
```

When analyzing Catherine's pseudocode, we are able to find lines that directly and clearly correlate with concrete lines of code. Lines B, C, G, and H can be easily converted to specific lines such as sum = 0 or sum + = 0. Line F is interesting because it accomplishes two things at once: iterating through each letter in the string and the parsing of each letter into an integer. This allows the student to ignore the decomposition of the string to integer sub-problem. She chose to go with pseudocode in the first place because of unfamiliarity with parsing input and we find that this decision enables her to avoid it entirely. We attribute this to her perception of the afforadances that come with pseudocode; earlier in the interview Catherine remarks that "The nice thing about pseudocode is that you don't see it, so the for loop is in that method. (points to pseudocode method)" The current pseudocode state terminates at the end of the string input missing the termination condition of 99. The state modifies the algorithm input from a list of integers into a list of strings only to be converted back into integers.

3.2 Episode 2 - Sylvia

Sylvia was similar to Catherine in coming up with a solution relatively quickly to first rainfall variation but had difficulty adapting their solution to the second variation. Unlike Catherine, Sylvia used recursion for her solutions to rainfall.

Using the break condition as base case which returns the average of a global list and the recursive step as appending the input to the global list if it is greater than 0. Recursion made transforming her solution difficult because she attempted to use recursion to parse the input from numbers to digit. This resulted in nested recursion that passed parameters into one function that tried to simultaneously solve both problems of parsing and rainfall.

Listing 4: Sylvia's Pseudocode Solution

```
a Rainfall (123,23,1):
b # no integers above 100
c if input == 99:
d   return avg(L)
e if input > 9:
f   append ( (modulo input 10) (L))
g if input < 0:
h   call.rainfall
i   Rainfall (modulo input 10)
j else:
k   append(input,L)
l   call.Rainfall
```

Two specific issues arise: (1) in lines F when appending the result of taking the input mod 10, we do not change the state of the input and thus always skip the condition in line G. This automatically results in a jump to line J where we append the original input into the list and then make a request to the user to pass in a new value. The program thus terminates after the user enters a value greater than -1. The second critical issue (2) occurs in line I where the result of our input mod 10 is passed into rainfall without storing any digit past the ones place. Line B refers to the student's explicit decision to change the original problem's scope: "Well I'm gonna put a limit, and assume that we're never going to have any integers above 100 because I don't feel like dealing with that. If you put like a million that's just rude."

3.3 Episode 3 - Tasha

Listing 5: Tasha's Part A Pseudocode Solution

```
a   rainfall(x)
a2  rainfall(xxxxxxx)
b       count = 0
c       sum = 0
d       prompt user for value (x) <- line 1
e       if x < 0:
f               return to line one
g       if  0 < x < 99:
h               count =  count + 1
i               sum = sum + x
i2              sum = sum + x + x + x + x + x + ...
j               return to line 1
k       if x > 99:
l               count =  count + 1
m               sum = sum + x
n               return to line 1
o       if x = 99:
m               return sum / count
```

Tasha used a low-level programming statement (what would be jump in Assembly) in place of a for loop and recursion to handle a continuous stream of input. This was an acceptable solution because we allowed the students to choose any language of their choice and students conception of pseudocode often led to producing a melting pot of languages.

	Dialogue
1	yeah so I don't exactly know like how it would like do this
2	like the right syntax or anything but
3	like instead of adding like x
4	I would just add like the digits of x I don't know
5	so like if this (participant pointing pencil at "prompt user for value (x)")
6	had like a lot of different like
7	it was just like 'x' 'x' 'x' 'x' 'x' 'x' (participant writes 4 x's next to the x in "prompt user for value (x)")
8	or something like that
9	then it would just be like this (participant writes "+ x + x + x + x + . . ." next to "sum = sum + x")

When Tasha is presented with the bonus they change their original solution by replacing the input value ”x” with ”xxxxx....” and replacing the ”sum = sum + x” with ”sum = sum + x + x + x + x + x + ...” The student's perception of pseudocode enables them to abstract the details of the Num-to-Digits design pattern (lines 1, 2). They add ellipses after the x's to tackle the traversal design pattern and general solutions (code line i2).

3.4 Episode 4 - Davida

In episode four we take snippets from Davida's interview to interrogate some of the reasons that students struggle with writing pseudocode.

	Dialogue A
1	It's weird to write out code
2	it doesn't feel natural
3	and I can't test it.
4	So I feel like it's all wrong
5	and it looks weird to see it in my handwriting.
6	And it its weird to not have things in color to say what they are.
7	And if something's not colored I'd always know oh I didn't call that right, but not here.
8	And I feel like not being told a language gives me so much freedom.
9	But I feel like if I was limited to a language I would not be able to act on it right now.

Davida describes writing code as weird and feels unnatural (lines 2, 3). Despite relating pseudocode to freedom (lines 8) she struggles with the lack of syntax color highlighting (lines 5-7) and code testing.

	Dialogue B
1	I know if that if I was actually typing all this code out there would be a lot of trial and error.
2	But on paper you get to assume everything works.
3	I also don't remember the perfect syntax, I just know that it's 'i plus plus'
4	(more speaking and writing code in between)
5	This would be so much easier if I could see what the errors popped up.
6	Or like what the input of my thing would actually be.
7	Like having what are they called, test files? Life savers in CS.

An Integrated Development Environment's (IDE) syntactic highlighting gives the user feedback on whether they are calling functions correctly, using objects correctly, and whether the scope of a call is legal. She remarks that seeing the errors returned by the IDE would make the task easier (line 5 in B).

	Dialogue C
1	I feel like syntax is really difficult to learn, it's like learning another language.
2	And I think that learning another language is tricky
3	cause I'm very much so adept at making formulas but
4	I feel like because we're splitting the same thing into like:
5	You learn python one semester, you learn java the next and then you learn like C++ or something.
6	It's like you never really perfect a language you just learn how to say the same thing in multiple languages.

Davida in another interview question elaborates on how pedagogy contributes to her being uncomfortable with writing pseudocode. She remarks that she feels as if she does possess a deep proficiency in any one language because her computer science courses up to this point have all be in different programming languages (lines 4 -6 in C). This contributes to her difficulty with recalling syntax and motivates her to use pseudocode where she can mix what she does remember from different languages.

4 Conclusion

From our study, we conjecture that even though pseudocode quality has been shown in previous research to have a positive correlation with answer quality, there can be unforeseen and sometimes negative consequences of writing pseudocode. Pseudocode that is poor in quality because of student misconceptions and lack of experience with a pseudocode first pedagogy model instead of code first can be detrimental in the problem decomposition process. We hope that our findings aids faculty who teach pseudocode first and emphasis program design before iteration.

References

[1] Belinda Copus and W. Perry Copus, Jr. Pseudocode quality correlations with ultimate answer quality in cs1. *J. Comput. Sci. Coll.*, 33(5):145–150, May 2018.

[2] Juliet Corbin, Anselm Strauss, and Anselm L Strauss. *Basics of qualitative research.* sage, 2014.

[3] Randi A Engle, Faith R Conant, and James G Greeno. Progressive refinement of hypotheses in video-supported research. *Video research in the learning sciences*, pages 239–254, 2007.

[4] Colin Fidge and Donna Teague. Losing their marbles: Syntax-free programming for assessing problem-solving skills. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95*, ACE '09, pages 75–82, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc.

[5] Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '05, pages 14–18, New York, NY, USA, 2005. ACM.

[6] Orna Muller, David Ginat, and Bruria Haberman. Pattern-oriented instruction and its influence on problem decomposition and solution construction. *SIGCSE Bull.*, 39(3):151–155, June 2007.