Recursion or Iteration: Does it Matter What Students Choose?

Ramy Esteero
University of Toronto Mississauga

Mohammed Khan
University of Toronto Mississauga

Mohamed Mohamed
University of Toronto Mississauga

Larry Yueli Zhang
University of Toronto Mississauga

Daniel Zingaro
University of Toronto Mississauga

ABSTRACT

Recursion and iteration are two key topics taught in introductory Computer Science. This is especially so for CS2 students, as CS2 is the course where recursion is typically taught and where control-flow concepts are solidified. When asked to solve a problem that could feasibly be solved with recursion or iteration, what do CS2 students choose to do? And how does this choice relate to the correctness of their code? This paper provides one answer to these questions through an analysis of student exam responses to a problem on finding deepest common ancestors in trees.

We find that 19% of students choose to use iteration, 51% choose recursion, and 16% choose to combine both iteration and recursion. In terms of correctness, we find that students who choose iteration perform better than those who choose recursion and the combination of both. Additionally, we find concern in the number of students who seemingly do not understand what the question is asking. We end the paper with some comments on helping students choose an appropriate control-flow strategy and a discussion of this type of question on a final exam.

CCS CONCEPTS

• **Social and professional topics → Computer science education**;

KEYWORDS

CS2, novice programming, recursion, iteration

ACM Reference Format:
Ramy Esteero, Mohammed Khan, Mohamed Mohamed, Larry Yueli Zhang, and Daniel Zingaro. 2018. Recursion or Iteration: Does it Matter What Students Choose?. In *SIGCSE '18: The 49th ACM Technical Symposium on Computing Science Education, February 21–24, 2018, Baltimore , MD, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3159450.3159455

1 INTRODUCTION

Recursion is a core problem-solving technique and an important component in many CS2 courses. It is also famously challenging to students (though see [7] for a contrary perspective), leading to hundreds of papers on how [1], when [12], and why to teach recursion [5]. Iteration is typically taught before recursion and is often deemed easier to learn and apply [5, 11]. For many simple numeric or list-based problems, iterative solutions suffice, and recursive solutions do little more than simulate control-flow through a loop. By contrast, some problems on binary trees may be easier for students to solve by recursion (e.g. preorder traversal) and others may be easier to solve by iteration (e.g. perhaps searching a binary search tree).

When given an exam problem on binary trees, with no prespecified solution plan, what do CS2 students do? Do they fall back to more familiar iterative code? Or do they presume that the topic (binary trees) means that the problem is more amenable to a recursive solution, and write recursive code? Is there a relationship between this choice and performance on the question or the exam as a whole? Do students sometimes combine recursion and iteration? And what do typical solution attempts look like in each of these categories? The present paper provides initial answers to these questions through a large-scale quantitative analysis of CS2 student exam data.

2 LITERATURE REVIEW

There is a vast corpus of research on teaching and learning recursion (for a review, see [5]). Broadly, students struggle to learn recursion, in part due to the many ways that recursion can be understood and misunderstood. For example, one work [4] offers evidence of four techniques that students use to successfully trace recursive code: simulating stack-based execution, accumulating pending results, bottom-up substitution of the results of "smaller" recursive calls, and predicting without tracing at all. At the same time, students exhibit a variety of mental models of recursion, many of which are not viable [2]. For example, analyses of recursive traces on exams suggest that many students conceptualize recursion as a loop rather than a series of independent function instantiations, or as an evaluation that stops as soon as a base case is reached (i.e. there is no unwinding of the stack) [2].

It appears that students have particular difficulty writing recursive code on binary trees. One technique for classifying these difficulties is goal-plan analysis: the problem is broken down into its functional components (the goals), and students' solution attempts (the plans) are analyzed for each goal. In one study [8], students were asked to write a recursive function to count the number of nodes in a binary tree that have exactly one child. Only 4 of the 18 students successfully solved the problem. Furthermore, over half of the students added extra, unnecessary conditionals to the code to avoid recursive calls on simple trees. This "arm's-length recursion" generally led to increased code-length and additional errors. As predicted by earlier research [3], base cases proved particularly problematic for students. The authors' sample size was small (only 18 students), perhaps due to the effort required to manually analyze

ACM ISBN 978-1-4503-5103-4/18/02…$15.00
https://doi.org/10.1145/3159450.3159455

each submission. Our approach differs in that we use automated tests for a general picture of correctness, trading onerous goal/plan analysis for large-scale quantitative results.

When given the choice of writing recursive or iterative code, what do students choose? One study of CS1 students [11] suggests that students are more likely to choose iteration. Interestingly, students who wrote recursive code were more likely than students who wrote iterative code to correctly solve the problems. Further, students were able to comprehend sample recursive and iterative solutions at comparable rates, suggesting that student choice can be at odds with actual outcomes. The four tasks given to students in that study — adding numbers, factorial, Fibonacci, and generating permutations — all focus on numeric inputs. It is unclear to what extent the findings generalize to problems on recursive data structures.

Another study focused on student comprehension of recursive and iterative code on linked lists [6]. The authors found no significant difference between correctness of student explanations for recursive and iterative code that searches a linked list. However, when asked to explain code that copies a linked list, students performed significantly better when given the iterative code compared to the recursive code.

This prior work suggests that whether iteration or recursion is the "right choice" for students depends on the domain of the problem. To our knowledge, however, none of that work investigates what happens when students are allowed to choose their own strategy for solving a nontrivial problem on trees.

3 METHOD

3.1 Context

This study was conducted in the Jan-Apr 2017 CS2 course at a major North American research university. Students had learned Python in the prerequisite CS1 course. The CS2 course uses Python to introduce the following topics: object-oriented programming, exceptions, stacks/queues, recursion, binary and n-ary trees, linked lists, sorting, and runtime complexity. Students attend weekly labs, write two term tests and a final exam, and complete two assignments (one on recursion and one on trees). Recursion is a core focus of the course: students learn recursion in the context of numbers and strings, and then study recursion again when introduced to linked lists and binary trees. Students are shown examples of both recursive and iterative code on linked lists and binary search trees. For example, students study recursive code for tree traversals, but study iterative code for finding the minimum or maximum element in a binary search tree.

Of interest in this study is one question from the April 2017 final exam. The question provides the definition of a binary tree node class (named `BTNode`) that stores the following attributes:

- `self.item`: the item stored in the node
- `self.left`: the left child of the node
- `self.right`: the right child of the node
- `self.parent`: the parent of the node
- `self.depth`: the depth of the node in the tree

Then, the question gives the definition of the *common ancestor* of two nodes, and describes a Python function, `deepest_ancestor`, that returns the node that is the maximum-depth common ancestor of two given nodes `node1` and `node2`. Part A of the question asks for two example test cases for the function, and Part B asks students to write the Python code itself.

Our sample solution for the problem was an iterative one that uses two sequential loops. The first loop traverses up the tree from the deeper node until we reach the same depth as the shallower node; if both nodes are initially at the same depth, the loop does nothing. Following that loop, we have two nodes `m` and `n` that are at the same depth. The second loop then moves both `m` and `n` up the tree until they become references to the same node; this is the deepest common ancestor of the original two nodes. We did not write or expect a recursive solution to this problem, as there is no backtracking required. (Compare to traversing a binary tree, where each choice-point must be stored so that the other side of the subtree can be explored later.)

However, we observed while grading that a substantial portion of the students were using recursive approaches, or mixed approaches with both recursion and iteration. Their solutions also exhibited varying levels of correctness and code complexity. These observations inspired us to perform a close investigation of the patterns in their solutions.

We define the **depth** of a node as the number of edges required to traverse from the root to that node; the depth of the root itself is 0. Students were very familiar with binary tree nodes that stored an item, left reference, and right reference. Parents of nodes and node-depths were familiar to students as well, but were not included in typical Python nodes that were studied in the course. The reason for making them available in the exam question was so that students could use these references without computing them; such "noise" would have obscured progress toward the focus of the question (finding a deepest common ancestor).

3.2 Data Analysis

We collected 397 student exams and transcribed each answer to the deepest ancestor question into its own Python file. We then created 10 test cases covering different aspects of the problem (see Section 5.1 for more details). We use the number of passed test cases as a measure of correctness.

We also recorded, for each student, the total grade on the exam, and the grades on Part A and Part B of the question. The grade for Part A (out of 2 marks) is an indicator of whether the student fully understands the problem and what the question is asking. We take the total grade on the exam (out of a maximum of 50 points) as a general measure of how well the student grasps the knowledge learned in this CS2 course.

Each student's solution was manually inspected by the authors and placed into one of the following four categories:

(1) Inadequate: solution that used neither recursion nor iteration. This category also includes blank solutions.
(2) Recursion: solution that uses any recursion without using any iteration. This includes using recursion in a helper function.
(3) Iteration: solution that uses any type of iteration (for loops, while loops) without using any recursion. This includes using iteration in a helper function.
(4) Mixed: solution that uses both recursion and iteration.

4 RESEARCH QUESTIONS

Our research questions are as follows.

- RQ1: what is the proportion of submissions that fall into each of the four categories? Answering this question provides insight into the cues that students may use to decide which strategy (recursion or iteration) to pursue.
- RQ2: how does the choice of category relate to the correctness of the solution?
- RQ3: what is the relationship between choice of category and grade on the rest of the exam? RQ2 and RQ3 together provide data on whether one solution strategy is "easier" or more natural for students, and whether the recursion/iteration choice is made differently depending on student ability.
- RQ4: within each category, what are the common solution patterns and errors?

5 RESULTS

5.1 Test Cases

Part B of the question was graded out of 4 points by the course Teaching Assistants (TAs) as part of grading the exam. For a more fine-grained analysis of submissions, however, we decided to design test cases that could be run on each submission rather than use the TA grading. Tests were written in a style that is typical when an assignment is automatically graded by grading software. That is, we wrote each test case to target what we expect to evoke a different execution behaviour in at least some of our students' submissions. The test cases are as follows:

(1) node1 and node2 are the same node in a tree of a single node
(2) node1 and node2 are the same node in a larger tree
(3) node1 and node2 are distinct but on the same level of the tree, and the deepest common ancestor is the root
(4) node1 and node2 are distinct but on the same level of the tree, and the deepest common ancestor is not the root
(5) node1 is one level deeper than node2, and node2 is the deepest common ancestor
(6) node2 is one level deeper than node1, and node1 is the deepest common ancestor
(7) node2 is deeper than node1, and the deepest common ancestor is the parent of node1
(8) node2 is several levels' deeper than node1, and node1 is the deepest common ancestor
(9) node1 is several levels' deeper than node2, and node2 is the deepest common ancestor
(10) node1 and node2 are different nodes in an unbalanced tree

A desirable property of these tests would be that each test is passed by different numbers of students. This would suggest that the tests are able to discriminate among student solutions. Indeed, we see from Table 1 that the number of students passing each test varies widely.

We sequenced the tests in terms of what we perceived to be the difficulty of students passing the test; that is, we expected more students to pass the early tests and fewer students to pass the later tests. The tests do trend this way, but with considerable deviation. For example, Test 2 proved quite troublesome, as only 23% of students passed this test. We saw this test as a simple check that base cases were being handled properly. The fact that more students pass Test 1 than Test 2 suggests that the base case was interpreted by some students to be a single-node tree, rather than two references to the same node in an arbitrary tree. Comparing Test 4 to both Test 5 and Test 6, we see that students generally handled nodes on the same level better than nodes on different levels. Extending this reasoning, it is clear why students struggled in particular with Test 8 and Test 9: those tests are like Test 5 and 6, but have the added complexity of moving one node up multiple levels before advancing both nodes to their common ancestor.

It appears, then, that our test cases are meaningfully measuring student progress on the question. If question performance correlates with performance on the rest of the exam, then we would additionally expect a relationship between the number of tests passed and student exam grade. Table 2 demonstrates in general that the number of test cases passed correlates with the grade on the exam. (Note that the exam grade here is on the exam minus this question; otherwise, any relationship would be artificially increased as test cases passed is related to performance on the question.)

Table 1: Number of students who passed each test case.

Test Case #	Number of Students
1	115 (28.97%)
2	93 (23.43%)
3	140 (35.26%)
4	205 (51.64%)
5	88 (22.17%)
6	94 (23.68%)
7	116 (29.22%)
8	62 (15.62%)
9	61 (15.37%)
10	107 (26.95%)

Table 2: Relationship between number of test cases passed and exam grade.

# of Cases Passed	Average Exam Grade (%)
0	44.93
1	51.84
2	55.39
3	51.61
4	57.45
5	56.64
6	52.93
7	61.36
8	74.25
9	69.70
10	75.05

5.2 RQ1: Category Distribution

The data in Table 3 answers RQ1. We see that 14% of students provide solutions that use no iteration or recursion at all. Most of these submissions were blank or contained a few lines of code

Table 3: Submission counts and average correctness per category.

Category	Attempted By (397)	Average Test Case Grade (%)	Average Final Exam Grade (%)
Inadequate	56 (14.11%)	5.2	34.96
Iteration	74 (18.64%)	40.4	57.84
Recursion	204 (51.39%)	27.1	54.26
Mixed	63 (15.87%)	31.7	62.64

that checked some base cases (such as node1 and node2 having the same parent). Easily the most dominant category here is recursion, with over 51% of students choosing a recursive approach.

5.3 RQ2: Average Question Correctness

A majority of students chose to solve the question using recursion. But was this a good choice? Perhaps not, as shown in the "Average Test Case Grade" column of Table 3. This column contains the average correctness, per category, on the test cases. Those who used recursion averaged 27% on the test cases; those who used iteration performed more strongly, at an average of 40%. In fact, those who used iteration received the highest grades on this question. Unsurprisingly, those who did not use any iteration or recursion received the lowest grades. It is interesting to speculate on the "Mixed" group: those students who used both recursion and iteration. Their 32% correctness sits between the recursive correctness and iterative correctness. Why? One might suspect, as we did initially, that the "Mixed" solutions would be erroneous due to inappropriate combining of iteration and recursion. While this certainly did happen, other solutions were "Mixed" in the sense that recursion was used for one task and iteration for another. We explore this further in Section 7.

5.4 RQ3: Average Exam Correctness

Revisiting Table 3, we see that it is the students in the "Mixed" category that perform best on the rest of the exam. Unsurprisingly, the "Inadequate" solutions were associated with the lowest grades on the exam. Students in the "Recursion" category did not perform as well as those in the "Iteration" category.

A more fine-grained analysis of the relationship between category and exam grade is provided in Table 4. We have divided students along grade boundaries on the exam: 0-49 (F), 50-59 (D), 60-69 (C), 70-79 (B), 80-89 (A- and A), and 90-100 (A+). For each of these grade ranges, we calculate the percentage of students that both fell in that range and whose code on the question fell into the given category. For example, from the first row we see that 10% of students both received a 0-49 grade and provided an "Inadequate" solution to the question. One of our initial hypotheses in this research was that weaker students would "apply recursion by default" to a question that involves trees. This does not appear to be the case. Rates of recursion use were rather constant within each row, with the notable exception of the 90-100 students. Those A+ students did not use recursion at all. A small sample in the A+ range of the class limits generalizability, but we find it interesting that it is those top students that both produced non-recursive code and did so correctly.

Table 4: Relationship between category and exam grade.

	Category (% of 397 Students)			
Grade (%)	Inadequate	Iteration	Recursion	Mixed
0-49	10.33	6.30	20.91	2.52
50-59	2.27	3.27	9.82	3.78
60-69	0.756	3.53	8.06	3.53
70-79	0.756	3.53	9.07	3.78
80-89	0.00	1.51	3.53	1.01
90-100	0.00	0.504	0.00	1.26
Total	14.11	18.64	51.39	15.87

5.5 RQ4: Solution Patterns and Errors

In this subsection, we offer a few representative examples of the Python code that students wrote as the answer to Part B of the question, for the purpose of better understanding common misconceptions and errors.

5.5.1 Iterative Solutions. One efficient solution to the problem is to use simple loops starting from node1 and node2, tracing parent references until a common node is reached. However, even among the students who approached the problem using iteration, we did not find any instance that implemented this solution perfectly. A common pattern in the students' iterative solutions is a brute-force approach. The following piece of code is a typical example.

```python
def deepest_ancestor(node1, node2):
    # passing 10/10 test cases
    l1 = [node1]
    node1_s = node1
    while node1_s.parent:
        l1.append(node1_s.parent)
        node1_s = node1_s.parent
    l2 = [node2]
    node2_s = node2
    while node2_s.parent:
        l2.append(node2_s.parent)
        node2_s = node2_s.parent
    l3 = []
    for node in l1:
        if node in l2:
            l3.append(node)
    if node1 in l2:
        return node1
    elif node2 in l1:
        return node2
    else:
        d_lst = []
        for anc in l3:
            d_lst.append([anc.depth, anc])
            d_lst.sort()
    return d_lst[len(d_lst)-1][1]
```

The above code first finds all ancestors of node1 and all ancestors of node2 and stores them in separate lists. Then, it traverses those lists to find the deepest common ancestor. This solution is very inefficient but was written carefully and therefore passes all 10 test cases. This is an example of a common pattern in student solutions where they build their solutions "literally" from the description of the problem. In this example, the literal solution to the deepest common ancestor problem would be to first find all the common ancestors and then pick the deepest one.

The iterative submission below is one of the closest that we found to the idea of maintaining two node references, tracing up the tree until the correct node is found.

```
def deepest_ancestor(node1, node2):
    #passing 4/10 test cases
    if node1 and node2:
        a = node1.parent
        b = node2.parent
        if a.depth > b.depth:
            while a.depth > b.depth:
                a = a.parent
        if b.depth > a.depth:
            while b.depth > a.depth:
                b = b.parent
        while a != b:
            a = a.parent
            b = b.parent
        return a
```

Conceptually, the above code embodies the core idea for a simple and efficient iterative algorithm. However, it mishandles the base case by skipping over `node1` and `node2`, beginning instead at their respective parent nodes. This error caused this solution to fail the majority of the test cases.

5.5.2 Recursive Solutions. Below is a student solution that takes a recursion-only approach. It is conceptually very close to a correct solution. However, again, the base case is not covered correctly. If `node1` and `node2` are two references to the same node, then it returns the node's parent rather than the node itself. Overall, we observe that among the students who demonstrate a correct high-level idea of the solution strategy, missing the base case is a very common error.

```
def deepest_ancestor(node1, node2):
    #passing 4/10 test cases
    if node1.depth > node2.depth:
        node1 = node1.parent
        return deepest_ancestor(node1, node2)
    if node1.depth < node2.depth:
        node2 = node2.parent
        return deepest_ancestor(node1, node2)
    if node1.depth == node2.depth:
        if node1.parent == node2.parent:
            return node1.parent
        else:
            return deepest_ancestor(node1.parent,
                                    node2.parent)
```

5.5.3 Mixed Solutions. Recall that 16% of the student solutions involve both iteration and recursion. Submissions in this category typically include considerable code, often using additional memory or loops. Below is one example.

```
def deepest_ancestor(node1, node2):
    #passing 4/10 test cases
    def ancestor(node):
        if node.depth == 0:
            return []
        return [node.parent] + ancestor(node.parent)
    if node1.parent == node2.parent:
        return node1.parent
    node1_ancestors = ancestor(node1)
    node2_ancestors = ancestor(node2)
    for nod in node1_ancestors:
        if nod in node2_ancestors:
            return nod
```

This code represents the most common strategy among all "Mixed" solutions. The underlying idea is very similar to that of the example in Section 5.5.1 that uses the inefficient brute-force iterative approach. The only difference is that the present example uses a recursive helper function, rather than a loop, to obtain the list of ancestors of a node. The approach, if done carefully, could have passed all 10 test cases. However, the student who wrote the above example again makes the mistake of not correctly handling the base case where `node1` and `node2` are the same node.

5.5.4 Other Solutions. For the sake of completeness, there are also solutions that used neither iteration nor recursion. Unsurprisingly, these solutions do not work correctly beyond the base cases. Below is one such example. Interestingly, this code is yet again missing the base case of `node1` and `node2` being references to the same node, and ultimately fails all test cases.

```
def deepest_ancestor(node1, node2):
    #passing 0/10 test cases
    if node1.parent == node2.parent:
        return node1.parent
    elif node1.depth < node2.depth and \
            node1.parent == node2.parent:
        return node1.parent
    else:
        return node2.parent
```

6 ADDITIONAL OBSERVATIONS

To this point, we have focused on Part B of the question, the part in which students actually write the code to find the deepest common ancestor. However, we also had grades available for Part A of the question, which asked students to give two distinct examples for the behaviour of the `deepest_ancestor` function. As one such example, a student could have provided a single-node tree, specified that `node1` and `node2` both referred to that node, and specified that the correct node to return was that single node. TAs graded Part A as 0 (no examples provided, or examples incorrect), 1 (one correct example provided), or 2 (two correct, distinct examples provided). To receive a grade of 2, students were asked to provide two distinct examples; that is, supplying a single-node tree twice warranted a 1 rather than a 2.

Our interest here is in the relationship between grade on Part A, which we take as a measure of student understanding of the question, and number of test cases passed on Part B. The results are in Table 5. As an example of interpreting this table, consider the first row of data, corresponding to students who received a grade of 0 on Part A. 67% of these students passed 0 test cases; only 3% of these students passed all ten test cases. The rightmost column in this row shows that, on average, these students passed only 9% of the test cases. Contrast this to the students who received a grade of 2 on Part A. Only 30% of those students passed 0 tests, and on average passed 32% of the test cases. There is a clear relationship here between performance on Part A and performance on Part B. Unfortunately, we document that 58 students received a grade of 0 on Part A. That is, it seems that almost 15% of students failed to correctly interpret the question or, at best, failed to demonstrate their correct interpretation of the question through examples.

Table 5: Relationship between Part A grade and number of test cases passed.

	% of students passing test cases					avg % cases
Part A grade (2)	0	1-3	4-6	7-9	10	
0	67.24	25.86	3.45	0.00	3.45	9.31
1	45.45	32.73	10.91	1.82	9.09	21.82
2	29.93	33.80	17.96	4.58	13.73	31.94
Overall	37.53	32.50	14.86	3.52	11.59	

7 DISCUSSION

What have we learned about student use of recursion and iteration from this study?

First, a majority of students chose to use recursion on the question. This is coupled with the fact that recursive solutions were associated with lower grades on the question. It seems that many students therefore "chose the wrong approach" for the question. Perhaps students associate recursion with questions on trees, and therefore use recursion by default in these cases. Indeed, we spend considerable time in our CS2 course on recursion, and this is true of other representative CS2 courses as well [10]. Perhaps when faced with a "difficult" question, students assume that recursion is what they should use simply due to the context of a CS2 course. This suggests that instructors might explicitly highlight when and why recursion is used. For example, if there are no choice-points and no reason for backtracking, then recursion is unlikely to be useful. Further, recursion can sometimes complicate a solution: if the power of recursion isn't being harnessed, then it may simply add complexity for no benefit.

Second, we notice a considerable number of students who submitted "Mixed" code that used both recursion and iteration. In our experience, such solutions can be unnecessarily complex and therefore often incorrect. Indeed, we saw many such solutions. However, there were other submissions that made considerable progress. These submissions generally used recursion to traverse the tree to collect all ancestors into two lists (one for `node1` and one for `node2`), and then used iteration to find the deepest node that is common to both lists. This does demonstrate refinement of a large problem into two subproblems, but it is unnecessarily inefficient and difficult to implement completely correctly. Especially at the level of CS2, we argue that using loops and recursion together may generally indicate an unnecessarily complex solution. Students should think carefully about whether using both control-flow mechanisms in the same function is warranted.

Is our question a fair question for a CS2 course exam? It is surely a difficult question: it relies on several binary-tree concepts at once, augments the standard node with `depth` and `parent` references, and gives no guidance on whether students should use iteration or recursion. We nevertheless suggest that it is fair, as long as it is one of the more difficult code-writing questions on the exam. We argue for a mix of problem types, including code-reading and conceptual questions, with which to assess student learning [9]. Given the range of solution types exhibited by students, it may be worth asking questions on an exam about whether recursion or iteration should be used to solve a stated problem. Students could be asked several such questions without requiring them to code each solution, instead discussing the features of the problem that led to their choice. That is, we take seriously the notion that students' choice of control-flow structure is itself something to be cultivated and valued as an important CS2 outcome.

8 CONCLUSION

A typical CS2 course introduces recursion and spends considerable time helping students think recursively. And for good reason: the CS education literature clearly documents a wide range of student struggles with recursion [5]. Often, it is clear from a problem description whether recursion should be used in the solution. Does the problem involve trees? Does the problem seem like something that the instructor would claim is too difficult to solve iteratively? Does the problem involve $O(n \lg n)$ sorting? Perhaps students use such heuristics to decide whether recursion applies. Or, perhaps students simply assume that recursion applies because it is such an important part of CS2.

The analysis here suggests that students may not have a strong intuition for when to apply recursion and when to apply iteration. Practical teaching implications from this research include explicit guidance to students on how to choose control-flow structures, and having students choosing and justifying solution strategies on exam questions.

REFERENCES

[1] D. Ginat and E. Shifroni. Teaching recursion in a procedural environment — how much should we emphasize the computing model? *SIGCSE Bulletin*, 31(1):127–131, 1999.

[2] T. Götschi, I. Sanders, and V. Galpin. Mental models of recursion. *SIGCSE Bulletin*, 35(1):346–350, 2003.

[3] B. Haberman and H. Averbuch. The case of base cases: Why are they so difficult to recognize? student difficulties with recursion. *SIGCSE Bulletin*, 34(3):84–88, 2002.

[4] C. M. Lewis. Exploring variation in students' correct traces of linear recursion. In *Proceedings of the Tenth International Conference on Computing Education Research*, pages 67–74, 2014.

[5] R. McCauley, S. Grissom, S. Fitzgerald, and L. Murphy. Teaching and learning recursive programming: a review of the research literature. *Computer Science Education*, 25(1):37–66, 2015.

[6] R. McCauley, B. Hanks, S. Fitzgerald, and L. Murphy. Recursion vs. iteration: An empirical study of comprehension revisited. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 350–355, 2015.

[7] C. Mirolo. Is iteration really easier to learn than recursion for CS1 students? In *Proceedings of the Ninth International Conference on Computing Education Research*, pages 99–104, 2012.

[8] L. Murphy, S. Fitzgerald, S. Grissom, and R. McCauley. Bug infestation!: A goal-plan analysis of CS2 students' recursive binary tree solutions. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 482–487, 2015.

[9] A. Petersen, M. Craig, and D. Zingaro. Reviewing CS1 exam question content. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, pages 631–636, 2011.

[10] L. Porter, D. Zingaro, C. Lee, C. Taylor, K. Webb, and M. Clancy. Developing course-level learning goals for basic data structures in CS2. In *Proceedings of the 49th ACM technical symposium on Computer Science Education*, 2018.

[11] V. Sulov. Iteration vs recursion in introduction to programming classes. *Cybernetics and Information Technologies*, 16(4):63–72, 2016.

[12] S. Wiedenbeck. Learning iteration and recursion from examples. *International Journal of Man-Machine Studies*, 30(1):1–22, 1989.