Why Is Recursion Hard to Comprehend? An Experiment with Experienced Programmers in Python

Aviad Baron
aviad.baron@mail.huji.ac.il
The Hebrew University
Jerusalem, Israel

Dror G. Feitelson
feit@cs.huji.ac.il
The Hebrew University
Jerusalem, Israel

ABSTRACT

Recursion has the reputation of being hard to teach and understand. Our goal is to identify precisely what it is about recursion that makes it hard, and use this to devise a systematic teaching plan. We first make a distinction between regular recursion and tail recursion — the special case where the recursive call is the last command that is executed by the function. Tail recursion is the preferred form used in functional programming, because it simplifies memory management, and we hypothesize that it is also easier to understand. We conducted a controlled experiment with 139 participants, in which they were asked to understand different recursive functions. This revealed that tail recursion, when it is natural to use, is indeed easier to understand than the more general form of recursion where significant processing is performed after the recursive call. But it also showed that using tail recursion may come with a price, as when achieving the tail form requires a transformation of the code that obfuscates the underlying recursive algorithm. We conclude that having significant processing after the recursive call, or distorting the code so as to remove such processing, are major factors that make recursion hard. We therefore suggest to start teaching recursion with a basic form of tail recursion, and then progress to more complicated cases with processing after the recursive call. Transformations to tail form should be taught last, if at all.

CCS CONCEPTS

• **Theory of computation → Functional constructs**; • **Software and its engineering → Designing software**; • **General and reference** → *Experimentation.*

KEYWORDS

Recursion, Code comprehension, Computer science education

ACM Reference Format:
Aviad Baron and Dror G. Feitelson. 2024. Why Is Recursion Hard to Comprehend? An Experiment with Experienced Programmers in Python . In *Proceedings of the 2024 Innovation and Technology in Computer Science Education V. 1 (ITiCSE 2024), July 8–10, 2024, Milan, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3649217.3653636

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ITiCSE 2024, July 8–10, 2024, Milan, Italy
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0600-4/24/07…$15.00
https://doi.org/10.1145/3649217.3653636

1 INTRODUCTION

Recursion is a fundamental construct both in the theory of programming and for actually writing code. Recursion has been explored from various angles over the years. Several have highlighted the significant challenges involved in teaching and understanding recursion [10–12, 15, 17, 19–22]. They point out that recursion is "one of the most universally difficult concepts to teach" [7].

Some researchers compared the comprehension of recursive and iterative code, and tried to analyze the difficulties in understanding recursive code [2–4, 13, 16]. Benander et al. studied students' ability to comprehend recursive and iterative codes in Pascal [3]. The codes implemented searching in a list and list copying. Results showed that comprehension of the recursive search procedure was faster than that of the iterative code, with statistical significance. They conjectured the search task might be "more naturally recursive than the copy task; searching continues on the basis of what occurred in the previous step, whereas copying continues independently of what occurred in the previous step".

McCauley et al. replicated this study in Java [16]. Unlike the original study, the replication found that students could just as easily comprehend the recursive and iterative search methods. They suggest the difficulty students had in the earlier study with the iterative search method may have been due to the way the method was written in Pascal.

Several studies describe different mental models used by students when they learn recursion [8, 10, 12, 14, 20]. Specifically, students tend to understand recursion as a form on iteration, which is wrong. George named the flow of control leading to the recursive call the *active flow*, and the flow back from it the *passive flow* [8]. A common mistake is when students did not follow the passive flow at all, and simply calculated the solution at the base case [6, 9]. They apparently have the misconception that the base case stops all the calculation.

An important class of recursive functions are those that use "tail recursion". Tail recursion is defined as a recursive function in which the recursive call is the last command that is executed by the function. In other words, in tail-recursive functions there are no instructions after the recursive call. We believe that this may make them easier to understand. Tail recursion may also be more efficient than non-tail recursion, therefore in functional languages it is recommended to use the tail form.

Only few previous studies have focused specifically on comparing the understanding of tail recursion to non-tail recursion. This comparison can shed light on what makes recursion a challenging concept to students to understand. Therefor, we designed an experiment that deals with understanding different types of tail recursion versus non-tail recursion.

Our results indicate that tail recursion is indeed more readable when the embedded recursion contains many non-trivial commands after the recursive call. However, there are cases of recursion that formally is not tailed, but which include only one simple operation after the recursive call. In these cases, if one transforms the code to achieve tail form, this transformation makes the code harder to understand. Specifically, a common methodology to create tail recursion is to add an "accumulating variable" parameter to the function. This parameter is passed to recursive calls and serves as the place where the answer is built up. When the termination condition is reached, the final result is available in the accumulator and is simply returned with no further processing. But this is a somewhat artificial structure, and therefore harder to understand.

Our main contributions in this work are the following:

- Unlike previous studies, we examine understanding of recursion among experienced developers, and show that it is hard also for them;
- We compare understanding tail recursion and embedded recursion, and the implications of this comparison for the question of why recursion is difficult to teach and understand;
- We identify significant computation after the recursive call as an important factor in making the recursion harder to understand;
- We identify the use of additional parameters as an important factor that can make tail recursion difficult;
- We suggest that these results have implications for teaching recursion, and in particular the importance of presenting a wide variety of different examples in an order that helps build a full understanding of the concept of recursion.

2 RESEARCH QUESTIONS

Our work centers on the distinction between tail and non-tail recursion. But we do not expect one style to be globally preferable over the other. Our research questions therefore concern the factors that may affect which style is easier to understand:

(RQ1) Is tail recursion easier to understand than non-tail recursion? What factors have an effect on this comparison?

(RQ2) Does the use of an accumulating variable increase the difficulty of understanding tail recursion? Are base conditions in tail recursion with accumulating variables harder to write?

(RQ3) Can the use of an intermediate variable to separate the processing from the recursive call alleviate the difficulty of understanding recursion?

3 EXPERIMENTAL DESIGN AND EXECUTION

3.1 Experimental Materials

The experiment consisted of comprehension questions applied to various code snippets. The codes are all concerned with list processing using different nuances of recursion. In most cases these are native Python lists, and in one case regular linked lists.

All the code snippets were written for the experiment, taking several considerations into account. First, the code snippets should address the research questions. Second, they should not enable participants to deduce their functionality without really understanding the code itself. For this reason we do not give the functions meaningful names. Finally, they need to be short and avoid mixing different factors.

We used four groups of code snippets, implementing solutions to different problems. Three of these are 2×2 designs, where one dimension is using tail or embedded recursion, and the other dimension is one of the factors we want to check. The fourth concerns understanding code with or without an accumulating variable, which is relevant only for tail recursion.

COUNT APPEARANCES. The first group contains four implementations of finding certain elements in a list. In addition to comparing tail or non-tail recursion, they compare the possible use of intermediate variables. Specifically, in one pair of codes the identification of the elements is embedded in the recursive call, and in the other it is separated by using an intermediate variable. The tail version in both cases uses an accumulating variable. These codes are used for RQ1 and RQ3.

The following code snippet is the base case. It implements regular recursion without an intermediate variable:

```
1 def func(lst, x):
2     if len(lst) == 0:
3         return 0
4     return (1 if lst[0] == x else 0)
5            + func(lst[1:],x)
```

The following code is an example of the other cases. This one implements the version with tail recursion (the recursive call is the last thing in the function) and using an intermediate variable (called w) to identify list elements before the recursive call.

```
1 def func(lst, x, w=0):
2     if len(lst) == 0:
3         return w
4     current_count = 1 if lst[0] == x else 0
5     return func(lst[1:], x, w + current_count)
```

REVERSE LIST and REVERSE PY LIST. The second group are four implementations of reversing a list. In addition to comparing tail or non-tail recursion, they compare the possible use of a Pythonic list implementation or a regular linked list implementation. The tail version in both cases uses an accumulating variable. These codes are used for RQ1.

The first of the four code snippets, using regular recursion on a linked list, is:

```
1 def func(curr):
2     if (curr == None):
3         return curr
4     if (curr.next == None):
5         return curr
6     next = curr.next
7     rest = func(next)
8     next.next = curr
9     curr.next = None
10    return rest
```

The second, using tail recursion on a linked list, is:

```
1 def func(curr, prev = None):
2     if curr.next is None:
```

```
3           curr.next = prev
4           return curr
5       next = curr.next
6       curr.next = prev
7       return func(next, curr)
```

The versions using Python lists are shorter, as they do not need to manipulate pointers explicitly. The tail recursion one is:

```
1   def func(lst, result = []):
2       if len(lst) == 0:
3           return result
4       return func(lst[:-1] , result + [lst[-1]])
```

ALL NUMBERS. The third code group includes only two code snippets, both of which are versions of tail recursion. The functionality of both is to check if all elements in the list are numbers. One uses tail recursions where the results is accumulated in a special variable, and returned this variable when the termination condition was met. The other uses a different tail call optimization without the need for an accumulator variable. This is used for RQ2.

The code without the accumulating variable is:

```
1   def func(lst):
2       if lst == []:
3           return True
4       elif isnumber(lst[0]):
5           return func(lst[1:])
6       else:
7           return False
```

The code with the accumulating variable is:

```
1   def func(lst, result = True):
2       if lst == []:
3           return result
4       else:
5           return func(lst[1:],
6                       result and isnumber(lst[0]))
```

Base case completion. Finally, the fourth group has four incomplete code snippets, where participants are required to complete the base condition of the recursion. Two are tail recursion and two are non-tail recursion, with the tail versions using accumulating variables. Note that in both cases, the description of the functionality was presented with the question, and the participants were only asked to complete the base case conditions. These codes are used for RQ1 and RQ2.

The firs pair calculates the length of a list. The code for the embedded recursion version is the following, with ?? representing the part that the experiment participants were required to complete:

```
1   def list_len(lst):
2       if len(lst) == 0:
3           return ??
4       return 1 + len_list(lst[1:])
```

The tail recursion version is:

```
1   def list_len(lst, w = 0):
2       if len(lst) == 0:
3           return ??
4       return list_len(lst[1:], w + 1)
```

The second pair is the same: a embedded recursion and a tail recursion, where participants need to complete the base case. The only difference is in the problem being solved, which is summing the number of occurrence of numerical characters in all the strings in a list of strings. These codes are used for RQ1 and RQ2.

3.2 Experimental Task

The task presented to the experiment participants was to provide a short description of the functionality of the code snippets. This represents a higher level of comprehension than just saying what a code prints, which can be achieved by tracing the code without really understanding it.

The descriptions given by the participants were graded on a binary scale of "correct" as opposed to "incorrect". This was based on an agreed rubric of the minimal information that an answer should include.

3.3 Experiment Execution

The experiment was conducted online, using the Qualtrics platform. This platform supported all the features we needed: question selection randomization and questions order randomization.

We recruited the participants through various methods, including reaching out to university students who were simultaneously working in the industry, distributing the survey link within a WhatsApp group of developers, contacting industry colleagues, and using online reddit forums for programmers. A total of 139 participants took part in the study and responded to at least one question. Among them, 89 reported their gender, with 81 identifying as male and 8 as female. Additionally, educational backgrounds were reported by 82 participants, with 52 holding a BSc degree, 17 holding an MSc degree, 9 holding a PhD, and 4 indicating only a high school education. 85 reported their years of experience as follows: 23 had 0-2 years of experience, 51 had 3-10 years of experience, and 11 had more than 10 years of experience. This raises the question of whether results from experienced developers also apply to novices and students who are just learning about recursion. We believe that they are, because recursion is a relatively rarely-used construct, so even experienced developers most probably do not have much experience with recursion.

The experiment started with an introductory page explaining that the experiment is about comprehension of recursive code, and explicitly targeted developers with a background in Python to ensure that they are familiar with Python syntax. We also noted that the expected time to complete the experiment was 10 to 15 minutes. This time estimate was based on the performance of the pilot participants. Participation was completely anonymous, and no identifying information was collected. We received IRB approval to conduct an experiment involving humans. Informed consent was explicitly implied by moving on the the experiment questions.

Each participant received one snippet from each of the four groups described above, selected at random. After the code questions we added an explicit question about searching for data on Google. The goal of this question was to neutralize the effects of unfamiliarity with a specific concept. Participants who reported using Google were excluded from the analysis.

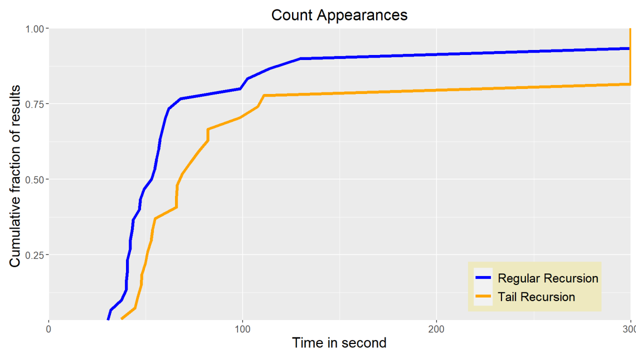

Figure 1: CDF of time for correct answers to COUNT APPEARANCES, tail (*N*=26) vs. non-tail (*N*=30).

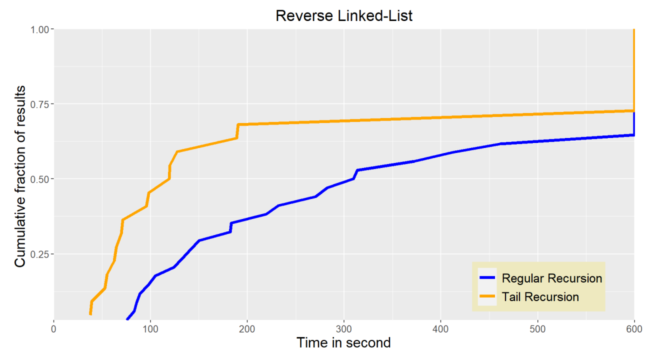

Figure 2: CDF of time for correct answers to REVERSE LIST, tail (*N*=22) vs. non-tail (*N*=33).

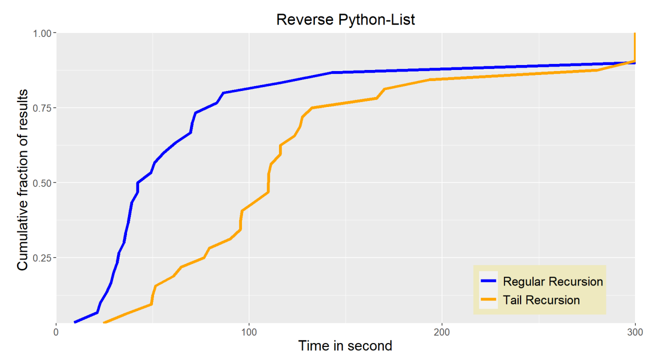

Figure 3: CDF of time for correct answers to REVERSE PY LIST, tail (*N*=32) vs. non-tail (*N*=31).

4 RESULTS

We have two types of results for each code snippet: correctness and time. The graphs below show the cumulative distribution function (CDF) of the time to correct results. The distributions are truncated at 5 minutes. This is a reasonable limit because the codes are short and simple, and are nearly universally understood in 1–2 minutes. We represent incorrect results by assigning them a time of infinity. As a result the CDFs converges to the fraction of correct results instead of to 1. In other words, one can see the fraction of incorrect results as the difference between the end of the graph and 1.

For example, the graphs in Figure 1 should be read as follows. The tail recursion version (the orange line) is more to the right than the embedded recursion version (the blue line). This means that the distribution of times is shifted a bit towards higher values. It also converges to a lower value of around 0.8. This means that there were around 20% wrong answers. The combination of these two observations indicates that the tail recursion version was harder to comprehend: it took more time *and* led to more errors.

4.1 Tail vs. Non-Tail Recursion

Commencing with the examination of RQ1, we delve into the comparison of tail-recursive and non-tail-recursive functions. To explore this aspect, we utilize various pairs of code samples. Figure 1 illustrates that, in the context of the COUNT APPEARANCES scenario, tail-recursive implementations exhibit approximately triple the number of errors compared to their non-tail counterparts, while also demonstrating slightly increased comprehension time. We interpret these findings to suggest that embedded (non-tail) recursion has a slight advantage over tail recursion when only a single uncomplicated operation follows the recursive call.

Different results are observed in the REVERSE LIST scenario. The results in Figure 2 indicate that this problem is harder than the previous one: it took more time and led to more errors. But in this case understanding the tail recursion is much faster than the embedded recursion. We conjecture that this is because changing the structure of the list requires many non-trivial operations after the recursive call. The reader of the code needs to keep this in mind to form a complete picture of what is happening, and this is hard.

This conjecture finds further support when comparing it with the second version of this problem, REVERSE PY LIST. In this case, the programming problem remains identical, but a Python list is used instead of a linked list. The results displayed in Figure 3 reveal that, in this case, the embedded version performs significantly faster. This discrepancy can be attributed to Python's abstraction of pointer manipulations, rendering the operations following the recursive call simpler and more straight-forward.

4.2 Using Accumulator Variables

The same set of results is applicable to RQ2, which pertains to the use of accumulating variables. In certain cases, such variables are necessary to transform a recursive function into a tail-recursive one. Specifically, the results in Figure 1 show that the tail version utilizing an accumulating variable exhibits slightly increased comprehension time.

A direct comparison between tail recursion with and without an accumulating variable is available in the ALL NUMBERS scenario. The results can be found in Figure 4. It's evident that the version with accumulating variables takes a longer time, corroborating the results mentioned above. There is no effect on correctness.

Using accumulator variable also has an effect on understanding the base case of tail recursion compared to the base case of embedded recursion. Understanding the base case has been recognized as indicative for understanding recursion [6, 21]. To check this we used two question where participants had to complete the base case

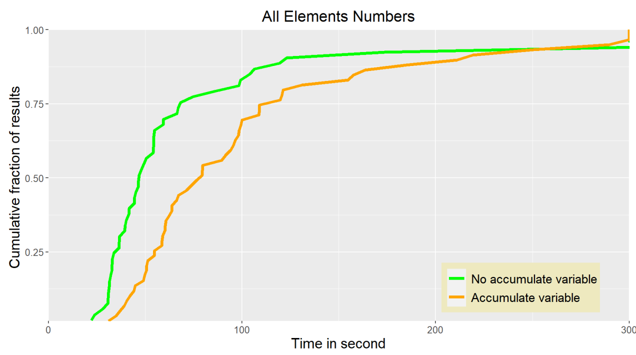

Figure 4: CDF of time for correct answers to ALL NUMBERS versions of tail recursion with (*N*=57) and without (*N*=53) an accumulating variable.

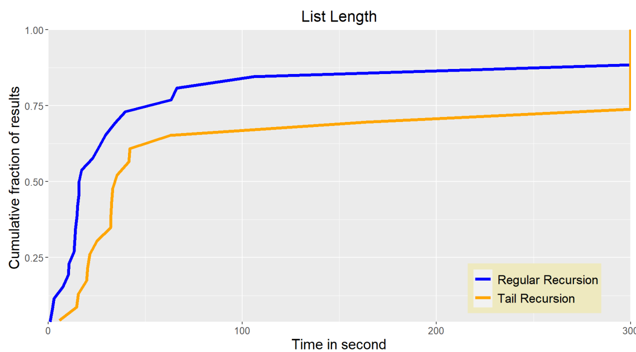

Figure 5: CDF of time for correct answers when completing the base case of calculating the length of a list, comparing tail (*N*=23) vs. non-tail (*N*=26) recursion.

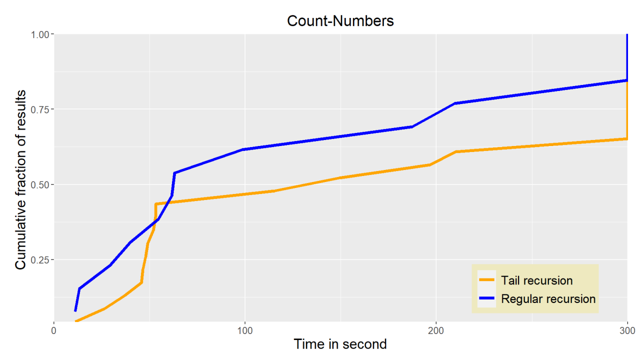

Figure 6: CDF of time for correct answers when completing the base case of counting numeric characters, comparing tail (*N*=22) vs. non-tail (*N*=15) recursion.

rather than understand a given code.

As can be seen in both Figures 5 and 6, the tail-recursive version was more challenging to complete in terms of timing and significantly more complex in terms of errors. So, in practice the base case in tail recursion is one of the challenging aspects of grasping

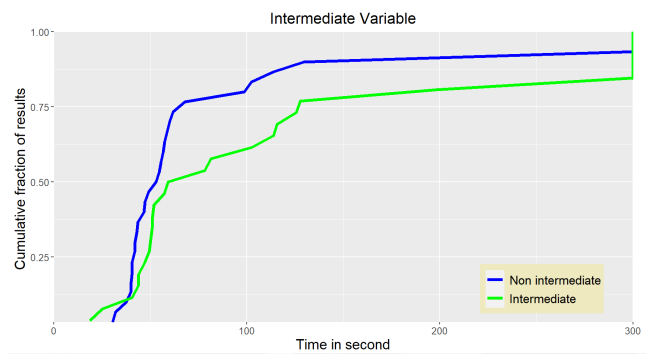

Figure 7: CDFs of time for correct answers for recursive code with (*N*=26) and without (*N*=30) intermediate variables. The results combine embedded and tail versions of COUNT APPEARANCES.

the concept, and appears to be less intuitive than the base case in non-tail recursion. This is perhaps because we are accustomed to thinking that the base case handles the smallest sub-problem, and, therefore, returns the answer to this small problem only. But when using accumulator variables, the base case actually returns the accumulated result of the whole calculation, not just of the smallest sub-problem, which may be confusing. This has implications for instruction and teaching recursive thinking, emphasizing the type of thinking required in this context and drawing attention to the fact that the base case can indicate the comprehension of the code in this context.

4.3 Using Intermediate Variables

In RQ3 we consider the use of intermediate variables to separate the processing of each element from the recursive call, which may make the code simpler to understand. Figure 7 shows that, at least in this case, it did not help. For half of the participants, and specifically those that reached a correct answer quickly, it did not matter whether the code used an intermediate variable or not. But for others the version with an intermediate variable took more time, or led to more mistakes in comprehension.

The reason may be that this case is actually rather simple, so adding code for the intermediate variable was detrimental rather than beneficial. Using intermediate variables may perhaps have a positive effect in more complicated cases, but this needs to be checked. A similar result, that intermediate variables only help in hard cases, was obtained by Cates et al. in the context of expressions unrelated to recursion [5].

5 THREATS TO VALIDITY

Construct validity. We measured the difficulty of understanding different recursive codes by measuring time and checking the correctness of the answers. This faces two threats. First, 'understanding' is not directly observable. We use a common proxy of explaining the functionality of code, which is very close (up to the need to judge the explanation and how closely it corresponds to the true functionality). Second, 'difficulty' is also not directly observable. We assume difficulty is reflected in time to solution and

in correctness, which are commonly used proxies [18]. However, there have been indications that they are not always correlated, because correctness may also be impaired by unmet expectations [1]. We therefore measure both, and in our case they are indeed correlated, thereby providing a measure of support for each other.

Internal validity. Internal validity is endangered when there are alternative explanations of the results. In our case, an example of such a risk would be using a functional language in which tail recursion elimination is an inbuilt feature. In such languages developers may be accustomed to solving certain cases with tail recursion, making it the natural choice and giving it an advantage over embedded recursion. To reduce this threat we conducted the experiments in Python, in which use of both tail and embedded recursion is not widely popular, making it less biased in favor of either approach.

The programmers we recruited have industry experience, which sets them apart from a specific paradigm they might have been exposed to in academic courses.

External validity. There are a lot of cases and examples of recursions and tail recursion. Our research examined only a limited number of basic examples. It's important to acknowledge that the results for the expression we employed may not necessarily generalize to other scenarios of embedded recursion, tail recursion, filter, reduce, etc.

6 IMPLICATIONS FOR TEACHING RECURSION

This research illustrates how recursion continues to be a complex issue even for experienced developers. Our observations can also make a significant contribution to the teaching of recursion. Prior researchers claimed that "Students need to be shown a wide range of recursive problems, particularly embedded recursive functions with commands that have to be executed after the recursive call" [20]. We agree, but augment this recommendation with a detailed suggestion of *the order* that concrete examples should be taught.

Both past studies and our study indicate that tail recursion in its basic form is the easiest to understand. It requires an understanding of the basic concept of recursion, but without additional difficulties. More advanced examples of recursion require an understanding that the calculation does not end when the termination condition is reached, and continues after the recursive call.

Based on our experiment we distinguish three levels. The first is when there are no additional commands after the recursive call, as in tail recursion. The second is where the recursion is formally not a tail recursion, because it has some processing after the recursive call, but this processing is trivial so it is easy to follow. Only the third is recursion that is more difficult to follow, with many non-trivial commands that are executed after the recursive call. This distinction has not been emphasized enough in previous work.

Based on the above considerations, we suggest that recursion be taught in four steps:

1. First, teach the basic idea of a function that calls itself. This includes the need for a termination condition. It should be done using the simplest examples possible, which are problems that are naturally solved using tail recursion. These are similar to iterative solutions, so should be easy to understand for novices [2]. An example is searching a list for a certain element.
2. Second, emphasize the importance of the passive flow, and the return to additional processing after the recursive call. It should be done with simple examples with trivial computation after the recursive call, which still resembles iteration and is therefore easy to follow. Calculating the factorial is the classical example. The non-tail recursive version of the COUNT APPEARANCES problem from our experiment can also be used.
3. Third, challenge the students with more complex forms of recursion, which require a more advanced mental model than iteration. An example is reversing a linked list as described above. This is a good example because the recursion is still linear, and the difficulty in monitoring lies in the fact that there are many commands that are executed after the recursive call. Additional example of such cases can be found in [20]. It is important to also teach multiple recursion with a constant number of calls (as when traversing a binary tree, or calculating the Fibonacci sequence) and when the number of recursive calls is not constant, including the case that the recursive call is in a loop. An example is to check if an input string (e.g. "schoolbus") can be segmented into sub words[1].
4. Tail recursion is important for functional programming, so transformations to tail form should also be taught. Our experiment showed that sometimes such transformations cause difficulties. Therefore emphasis should be placed on transformation methods and above all on the use of accumulating variables. Good examples from our experiment are ALL NUMBERS and the calculation of the length of a list.

Only by exposing students to a wide variety of recursions will it be possible to ensure that they develop the full cognitive model necessary for understanding recursion.

7 CONCLUSIONS

The comparison of tail recursion to embedded recursion exposes the computation after the recursive call as a factor that may make recursion hard to understand. Tail recursion resembles iteration, and aligns more closely with our intuitive mental models. In contrast, embedded recursion with a non-trivial computation after the recursive call introduces complexities that require a deeper, often less intuitive, mental model to navigate. But cases with a trivial computation after the recursive call, for example when the last instruction in the function is an expression that includes the recursive call, enjoy the cognitive advantage of tail recursion, in that you already know what the function does when you encounter the recursive call.

These observations suggest a natural progression for teaching the basic elements of recursion. Start with tail recursion to introduce the idea. Add a trivial computation after the recursive call to distinguish recursion from iteration. Only after these are established show the full force of recursion with more complicated examples. Our future work is to test this in a course setting.

The experimental materials are available at 10.5281/zenodo.10846692.

[1] e.g. https://www.geeksforgeeks.org/word-break-problem-dp-32/

REFERENCES

[1] S. Ajami, Y. Woodbridge, and D. G. Feitelson. Syntax, predicates, idioms — what really affects code complexity? *Empirical Software engineering*, 24(1):287–328, Feb 2019.

[2] A. Baron and D. G. Feitelson. How a data structure's linearity affects programming and code comprehension: The case of recursion vs. iteration. In *34th Workshop Psychology of Programming Interest Group (PPIG)*, Aug 2023.

[3] A. C. Benander, B. A. Benander, and H. Pu. Recursion vs. iteration: An empirical study of comprehension. *J. Syst. Softw.*, 32(1):73–82, 1996.

[4] A. C. Benander, B. A. Benander, and J. Sang. An empirical analysis of debugging performance - differences between iterative and recursive constructs. *J. Syst. Softw.*, 54(1):17–28, 2000.

[5] R. Cates, N. Yunik, and D. G. Feitelson. Does code structure affect comprehension? on using and naming intermediate variables. In *29th IEEE/ACM Intl. Conf. Program Comprehension*, pages 118–126. IEEE, 2021.

[6] D. Dicheva and J. Close. Mental models of recursion. *J. Educational Computing Research*, 14(1):1–23, 1996.

[7] J. Gal Ezer and D. Harel. What (else) should CS educators know? *Communications of the ACM*, 41(9):77–84, Sep 1998.

[8] C. E. George. EROSI - visualising recursion and discovering new errors. In *Proc. 31st SIGCSE Technical Symp. Computer Science Education*, pages 305–309, 2000.

[9] T. Götschi, I. D. Sanders, and V. Galpin. Mental models of recursion. In *Proc. 34th SIGCSE technical symp. Computer science education*, pages 346–350, 2003.

[10] J. E. Greer. A comparison of instructional treatments for introducing recursion. *Computer Science Education*, 1(2):111–128, 1989.

[11] B. Haberman and H. Averbuch. The case of base cases: Why are they so difficult to recognize? *Proc. 7th Conf. Innovation & Tech. in Comput. Sci. Education*, pages 84–88, 2002.

[12] H. Kahney. What do novice programmers know about recursion. *Proc. SIGCHI Conf. Human Factors in Computing Systems*, page 235–239, 1983.

[13] C. Kessler and J. Anderson. Learning flow control: Recursive and iterative procedures. *Human-Computer Interaction*, 2(2):135–166, 1986.

[14] N. Kiesler. Mental models of recursion: A secondary analysis of novice learners' steps and errors in Java exercises. In *33rd Workshop Psychology of Programming Interest Group*, pages 226–240, 2022.

[15] E. Lee, V. Shan, B. Beth, and C. Lin. A structured approach to teaching recursion using cargo-bot. In *10th Conf. International Computing Education Research*, page 59–66, July 2014.

[16] R. A. McCauley, B. Hanks, S. Fitzgerald, and L. Murphy. Recursion vs. iteration: An empirical study of comprehension revisited. In *46th ACM Tech. Symp. Comput. Sci. Education*, pages 350–355, 2015.

[17] C. Mirolo. Is iteration really easier to learn than recursion for CS1 students? In *Proc. 9th Conf. International Computing Education Research*, pages 99–104, Sept 2012.

[18] V. Rajlich and G. S. Cowan. Towards standard for experiments in program comprehension. In *5th International Workshop on Program Comprehension*, pages 160–161, Mar 1997.

[19] I. D. Sanders, V. Galpin, and T. Götschi. Mental models of recursion revisited. In *11th Innovation & Tech. in Comput. Sci. Education*, pages 138–142, Jun 2006.

[20] T. L. Scholtz and I. D. Sanders. Mental models of recursion: investigating students' understanding of recursion. In *15th Innovation & Tech. in Comput. Sci. Education*, pages 103–107, Jun 2010.

[21] J. Segal. Empirical studies of functional programming learners evaluating recursive functions. *Instructional Science*, 22:385–411, 1995.

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