How A Data Structure's Linearity Affects Programming and Code Comprehension: The Case of Recursion vs. Iteration

Aviad Baron
Dept. Computer Science
The Hebrew University
aviad.baron@mail.huji.ac.il

Dror G. Feitelson
Dept. Computer Science
The Hebrew University
feit@cs.huji.ac.il

Abstract

Iteration and recursion are two ways to traverse data structures. But when is one of them preferable over the other? Some researchers argue that it is natural to use recursion to process recursively defined data structures, such as linked lists and trees. In order to test this, we conducted two experiments of writing and understanding code segments related to the processing of different data structures, with about 100 participants in each. The results showed that the way the structure is defined is not a significant factor affecting the nature of the processing. On the other hand, the main factor that influenced the selected approach was having a "linear mindset", either due to the linearity of the structure (where data items are arranged in a linear fashion, as in a list), or the linearity of the processing path (where data items are visited in a linear manner, without branching). When we process a linear structure like a linked list, the nature of its processing tends to be more iterative; a similar but weaker effect is obtained when we process a linear path in a tree, for example when looking for an item in a binary search tree. But when we need to understand code, recursion is problematic in the event that the processing is divided both before and after the recursive call, and especially when the processing after the call (the so-called "passive flow") is non-trivial. These results provide insights into the cognitive factors which affect how programmers interact with code. They can be explained by the observation that linear processing and simple recursion (with a trivial or non-existent passive flow) are more suitable for the serial thinking of humans.

1. Introduction

Data structures are used to store and organize data. This organization often also reflects the way the data is processed. But does it *dictate* the way they are processed? In this study we examine how the topology of the structure affects its processing, and specifically the choice of recursive versus iterative processing.

Recursion is a basic construct in computer programming, where a function calls itself. A possible alternative is to use iteration, where a set of operations is performed repeatedly within a loop. In both cases this continues until a specific stopping condition is met.

The concept of recursion is also used in data structures. For example, a linked list is commonly defined recursively, in that each element contains a pointer to the remainder which is also a list. Thus a linked list is either:

- The empty list, represented by `None`, or
- A node that contains some data and a reference to a linked list

Given this definition, it seems natural to process such a list using a recursive function:

- If the list is empty, handle this case, or
- Handle the data in the head of the list, and then make a recursive call to handle the rest of the list

This approach is advocated by many, e.g. (McCauley, Hanks, Fitzgerald, & Murphy, 2015; Bloch, 2003; Choi, 2021; Pattis, 2006; Elkner, Downey, & Meyers, 2012; Snyder, n.d.). For instance, Turing Award laureate Niklaus Wirth wrote (Wirth, 1976, p. 127):

> "Recursive algorithms are particularly appropriate when the underlying problem or the data to be treated are defined in recursive terms."

But from our personal experience we observe that we often actually process linked lists using loops rather than using recursion. We therefore designed two experiments to check the relative advantages of recursion and iteration. The first experiment deals with *writing* code, and checks whether recursively defined data structures are indeed natural to process recursively — and also identifies other factors that may influence this choice. The second deals with *understanding* given code, which is written using either recursion or iteration. These experiments had 97 and 101 participants respectively, most of them from industry. Both are controlled experiment that compare data structures with different topologies, in particular linked lists and trees.

The code development experiment included questions about basic processing tasks on these data structures. Importantly, these tasks employ different paths for traversing the data structures. What we found was that the recursive definition of a data structure does not have a statistically significant effect on the processing method: for example, linked lists are indeed most often processed using iteration. The more important factor appears to be having a "linear mindset". If it is appropriate to think about the processing as a linear progression, there is an increased tendency to use iteration. This happens when the topology is linear (as in a linked list) or when the processing path is linear (as in a binary search tree). The effect is even stronger if the path is known in advance and does not depend on the input.

In the code comprehension experiment we try to identify different factors that may influence the understanding. This experiment led to more mixed results: in one case the recursive version was more understandable, in another the iterative version was more understandable, and in two there was no major difference. It seems that recursion has an advantage when it simplifies and shortens the code. But iteration again can have an advantage when it presents the processing in a more linear manner. For example, this may happen when we can avoid saving state when making the recursive call and then performing non-trivial operations on it on the exit path.

Our main contributions in this work are

- We focus on how developers in industry think about recursion, as opposed to studies on students in an academic setting;
- We design experiments based on scenarios of processing data structures, instead of classic academic examples that do not necessarily reflect real situations in programming life;
- We show that linear processing is more often done using iteration, especially if it is input-oblivious, even when the underlying data structure is defined recursively;
- We find that recursion can be hard to understand when it includes non-trivial computation after the recursive call.

In terms of practical implications, our results can inform technical debates on software design. For example, rather than accepting the dogma that recursive structures should be processed recursively, developers may benefit from taking the linearity of the processing and the length of the code into account. Likewise, our results suggest additional factors that may inform the debate about learning materials. Educators can use them to support learning by choosing examples that are both natural and simple.

2. Related Works

Recursion has been addressed in several studies on code comprehension and debugging, development, and more (Benander, Benander, & Pu, 1996; Kessler & Anderson, 1986). Benander et al. (Benander et al., 1996) compared iterative and recursive versions of search and copy routines, using student subjects and Pascal. Their experimental participants understood the recursive version of search better than the

iterative version, and this result was statistically significant. However, for copy they understood the iterative version better, but this result was not statistically significant. McCauley et al. (McCauley et al., 2015) replicated this study and found that the recursive version of search is not better than the iterative version. Their interpretation of the difference in results was the way the search code is written in Pascal vs. Java.

Several papers discuss cognitive models of understanding recursion. These first require some definitions. Recursion is defined as "a process that is capable of triggering new instantiations of itself, with control passing forward to successive instantiations and back from terminated ones" (Kahney, 1983). George (George, 2000) then defines *active flow* to mean when control is passed forward to new instantiations, and *passive flow* when control flows back from terminated ones. The main model of how recursion is understood is the "Copies Model" which allows programmers to accurately and consistently represent the mechanics of recursion. Other models may lead to misconceptions because they capture only part of what happens in recursion (Kahney, 1983; Kiesler, 2022; Scholtz & Sanders, 2010). As an example of such misconceptions, several studies have identified base cases as a source of error (Dicheva & Close, 1996; Segal, 1995). They found that participants tend to look at the stop condition as stopping all processing, and ignore the processing after the recursive call (the passive flow).

Researchers have also examined the pedagogical aspect, in order to find methods that promote a correct perception of the recursive concept (Kahney, 1983; Sanders, Galpin, & Götschi, 2006; Scholtz & Sanders, 2010). Aqeel et al. use eye tracking to study how students read iterative vs. recursive code, but do not find any significant differences (Aqeel, Peitek, Apel, Mucke, & Siegmund, 2021). Some researchers claim that recursion is a hard concept to teach (Haberman & Averbuch, 2002; Kahney, 1983). On the other hand, other researchers have wondered if this is really a difficult concept to teach, or perhaps the difficulty is due to the chronological order of teaching recursion versus iteration (Mirolo, 2012). Wiedenbeck pointed out the importance in teaching recursion by providing a wide variety of examples (Wiedenbeck, 1989). She performed an experiment in which she compared two groups of students, a group that was given a variety of examples and another group that was given only a few abstract examples. Students who learned from a variety of concrete examples solved more questions correctly. Other researchers have also dealt with the question of which materials and examples should be used to teach recursion. For example, some have suggested teaching recursion using mathematical formulas (Segal, 1995; Greer, 1989). Another approach talks about introducing recursion in a "real life" connotation. They suggest building a game using recursive code as an example (Lee, V., Beth, & Lin, 2014).

3. Experiment 1: Development

3.1. Research Questions

We are interested in knowing when programmers tend to use a recursive approach and when they prefer the iterative alternative. More specifically, we would like to test whether processing recursively defined structures is necessarily recursive. Our hypothesis is that the reality is much more complex. We would therefore like to locate cases where the tendency is not so unequivocal, and identify which factors naturally lead to implementing a programming task in a recursive way, and alternatively which factors or properties make a programming task to be processed naturally in an iterative way.

The world of programming is wide and diverse, and in this experiment, we try to capture specific characteristics. Our concrete research questions were:

RQ1. Are recursive data structures, like linked lists and trees, usually processed in a recursive manner?

RQ2. Are linear data structures, like linked lists and arrays, usually processed iteratively?

RQ3. Are linear processing trajectories, like in a binary search tree, usually processed iteratively?

RQ4. Is there a difference between linear processing in which the trajectory is known in advance and cases where it is not known in advance?

Table 1 – Summary of programming problems used in the first experiment.

Name	*Data structure*	*Description*	*RQ*
ARRAY	array	check identity of two arrays	control
LIST	linked list	count positive elements	RQ1 RQ2
FACTORIAL	–	compute the number of possible permutations of different elements	RQ1 RQ2
TREE	binary tree	count odd elements	RQ1
SEARCH	binary search tree	check if a value exists	RQ3
MAX	binary search tree	find maximal value	RQ4

RQ5. If the choice between recursion and iteration dichotomous, or are there cases where both approaches are used?

3.2. Methodology

Our methodology is to conduct a controlled experiment to understand which approach (iterative or recursive) programmers take in different situations.

3.2.1. Experimental Materials

The experiment is based on data structures problems, that can be solved either iteratively or recursively. We use different data structures to see how their properties, and the specific programming problem, affect the choice of using iteration or recursion.

There were several methodological considerations behind the choice of problems. First, we wanted to avoid canonical problems, i.e., those that are typically taught in programming courses in academia or in other training courses. At the same time, we wanted the problems not to be too difficult. This had two reasons. First, we wanted participants to be able to answer the questions directly based on their knowledge, without resorting to Google. Second, we wanted the experiment to be reasonably easy, to keep it short and make it easier for us to recruit participants.

The programming problems we used in the experiment are outlined in Table 1. The considerations for choosing these problems are as follows.

1. ARRAY: Check the identity of two arrays. It is included as an example of a problem that is inherently iterative — the data structures are linear with no recursive structure, and it is natural to process them linearly using iteration.

2. LIST: Count the positive elements in a linked list. Like the previous problem this concerns a linear scan of a linear structure (the list), but in this case the data structure is *defined* recursively — each node in the list contains a reference to the remainder of the list.

3. FACTORIAL: Compute the number of possible permutations of an array with different elements. This is similar to the previous problem in that it deals with a linear expression that is naturally defined in a recursive manner. However, in this case it concerns a mathematical structure rather than a data structure.

4. TREE: Count the number of odd elements in a binary tree. This concerns a non-linear data structure which is defined recursively.

5. SEARCH: Check if a certain value exists in a binary search tree. This makes an interesting addition due to the fact that it combines a non-linear recursively defined structure with an algorithm that maps out a linear trajectory within this structure.

6. MAX: Find the maximal value in binary search tree. This is similar to the previous problem, but in this case we know in advanced the linear path in the tree — you always choose the right son.

All the problems together enable a controlled experiment, where the main factor that changes between treatments is the data structure. In ARRAY it is an array, which is a non-recursive linear structure. In LIST and FACTORIAL it is a recursive linear structure: either a linked list or a recursive linear computation. In TREE, SEARCH, and MAX it is a tree, which is a non-linear recursive structure. In TREE this is a general binary tree, and in SEARCH and MAX it is a search tree.

Together the problems are diverse and cover three dimensions. The first dimension is the recursivity of the structure. An array is not a recursive structure. But linked lists and trees are conventionally defined recursively, and have independent substructures which can be viewed as having the same features. The second dimension is the topology of the structure. Linked lists and arrays are linear structures, while trees have a non-linear structure. The third dimension is the predetermination of the traversal of the data structure. In linear structures there is only one option. And when searching for the maximum element in a tree we also know the direction of the route in advance. On the other hand, for finding a general element, different trees will require different trajectories, and this is only known at runtime.

Note that each problem can be solved either iteratively or recursively. All the problems are fundamentally similar and require a basic traversal of the structure. This reduces the number of confounding factors that may affect the results. At the same time, the differences between the problems mean that the same code or algorithm cannot be used for all of them.

The FACTORIAL problem is different from the others due to the fact that it does not deal with a data structure. We formulate the question in data structure terminology to be similar to other questions, and to obscure the fact that we focus on recursion.

The experimental task was to write code that solves these problems. Participants were not constrained which programming language to use, and indeed they used a diversity of languages (Python, C/C++, Java). As the code was typed into a Qualtrics questionnaire, there was no issue of compiling and testing.

3.2.2. 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. The description of the experiment was "an experiment on data structures". The purpose of this minimalistic description was to hide the true purpose of the experiment from the subjects. If we had called it "an experiment on recursion vs iteration" it would cause subjects to ponder this issue rather than doing what comes to them naturally.

At first, in order to estimate times, we conducted a small pilot study, and concluded that each subject should receive 3 problems. Each subject's problems were selected at random from the 6 problems listed above. Due to this randomization, the numbers of participants who received each problem were not identical. The order of the problems was also randomized, to neutralize the possibility that a problem on a particular topic that appeared previously will systematically affect the answer to a subsequent problem.

As far as the recruitment of participants is concerned, the general ambition was to locate a wide variety of subjects with different backgrounds. In many experiments on understanding and using code there is a tendency to use students, due to their accessibility and availability. Our ambition was to try to reach as wide an audience as possible, including both academics and professional developers. Specifically, we tried to avoid giving the questionnaire to first-year students as they would just do what they recently learned in class.

Participants were recruited in multiple ways. Some were recruited at the university, by approaching fellow students to advanced degrees and undergraduates in the programming labs. We also sent invitations to WhatsApp and Telegram groups of students. Others were recruited based on personal connections, including many that now work in industry. Participants were not compensated for their participation.

We received IRB approval to conduct an experiment involving humans. The experiment was preceded with an explanation and obtaining consent to participate. At the beginning of the questionnaire we gave

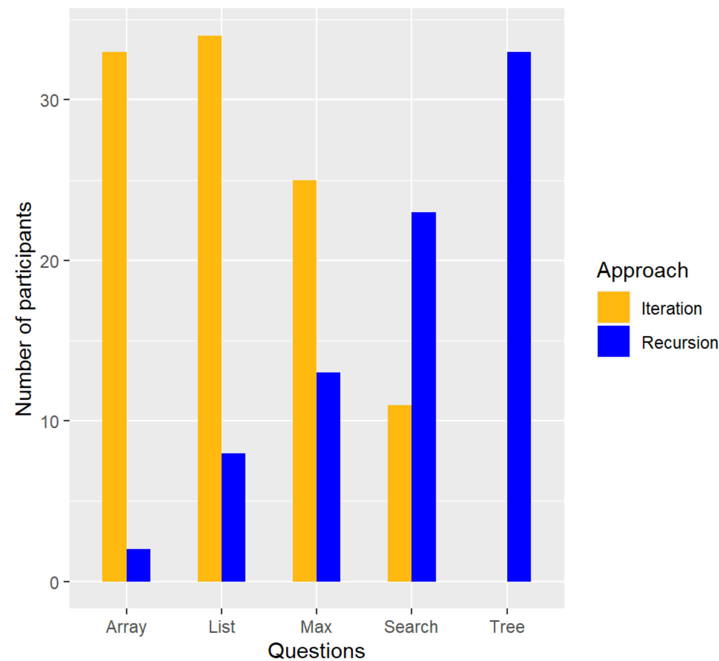


Figure 1 – Use of iteration and recursion in the different problems.

an introduction briefly recalling what the data structures are. In this introduction we tried to avoid an iterative or recursive description of the structure, but to give a neutral description as much as possible. At the end of the questionnaire we included demographic questions in which we collected data on years of experience, gender, etc.

3.3. Experimental Results

3.3.1. Participants and demographic background

The experiment was conducted from January to June 2022. All told, we had 97 participants who answered at least one problem. 85% of the respondents reported their gender: 82% of them were men and 18% women. In terms of education, 82% of the respondents reported their education: most of the participants had an academic background, but not all of them. 42 had a BSc degree or were studying for one, and 28 had an MSc or were MSc students. 7 were PhD students. We also asked about professional programming experience (excluding studies). 79% of them reported it. The median was 3 years. 28 participants had experience between 0–2 years and 49 had more than 3 years of experience.

3.3.2. Approaches for Traversing Data Structures

Basic Results

In total our participants provided 214 answers to the different problems. These answers were classified as using either iteration or recursion. Iteration was identified based on using `for`, `foreach`, or `while` loops. Recursion was identified based on calling the function from within the function itself.

Figure 1 shows the number of participants who elected to use iteration or recursion when solving the different problems. It is immediately obvious that the division between using iteration and recursion is not dichotomous: in most of the problems there was a mix of participants who used both approaches. This answers research question RQ5. However, one approach was typically preferred over the other.

The problems are listed in the graph in order of reduced use of iteration and increased use of recursion. For ARRAY the natural tendency by a wide margin is an iterative implementation: 33 participants used this approach, and only 2 used recursion. In LIST the common approach is still iterative processing, but the difference is somewhat less extreme, with 34 using iteration and 8 using recursion. At the other extreme, we see that in TREE the tendency for a recursive realization is universal: 33 participants used recursion and none used iteration.

The most interesting results occur in the remaining two problems, MAX and SEARCH. Both these problems involve searching in a binary search tree. And in both we observed relatively wide use of the two possible programming approaches. But the proportions were different. In MAX there is a somewhat greater tendency for iterative processing rather than recursive processing, with 25 participants choosing to use iteration and 13 using recursion. In SEARCH, on the other hand, the tendency is reversed: most of the participants, 23 of them, preferred a recursive implementation, while only 11 used iteration.

Statistical Analysis and Interpretation

Our experiments check the percentage of developers who use recursion in different programming situations. We would like to check whether the expected values of these percentages in different situations are unequal. For this we will use a between-subjects Z-test for proportions, where the null hypothesis is that they are equal, and the alternative hypothesis is that the expected values are different.

Research question RQ1 is whether a structure that is defined recursively is overwhelmingly natural to process recursively. In the experiment we checked this using a recursively-defined linked list (problem LIST). We then applied a statistical test to see whether the fraction of participants using recursion was above 50%. The results in Figure 1 indicate that this is not the case (only 8 of 42, which is 19.04%, used recursion), with an extremely high significance ($p < 0.0001$). This provides strong evidence against the claim that if something is defined recursively then it is natural to process it recursively.

To examine the degree of influence of the element of recursive definition on the way of processing we compared a data structure that is defined recursively (a linked list) and a data structure that is not defined that way (an array). We tested if there is a statistically significant difference in the expected values of the fraction of participants who use recursion and iteration in these two cases.

Formally, the independent variable is the way the data structure is defined (whether there is a tendency to define it recursively or not), and the dependent variable is the implementation mode (recursive or iterative). The null hypothesis is that expected value of the percentage of the participants who use recursion is equal in both cases. The alternative hypothesis is that the expected values of these percentages are different. The experimental results were that 5.7% used recursion in the ARRAY problem, and 19.04% in the LIST problem. According to the statistical test this difference is not significant, with a p-value = 0.1637. This means that the null hypothesis was not rejected. Again, we see that a recursive definition does not necessarily lead to recursive processing.

Therefore, the claim that a recursive definition is a significant factor influencing the processing, and specifically that it increases the tendency to process recursively, is not supported by our experiment. This is contrary to the claims made by some educators and researchers that it is natural to process a linked list recursively due to the tendency to define it recursively.

However, other recursive data structures are indeed processed recursively, as we saw for instance in the TREE problem. We suggest in RQ2 that this may be attributed to the topology of the structure: linked lists are inherently linear, while trees are branching. The statistical analysis shows that there is a statistically significant difference between the tendency to recursively process these two data structures. Formally, we define our independent variable to be the linearity of the data structure's topology, and the dependent variable to be the nature of the processing (recursive or iterative). The null hypothesis is that the expected value of the percentage of the participants who use recursion is equal in both cases. The alternative hypothesis is that the expected values of these percentages are different. The experimental results were that only 19.04% used recursion in LIST, and 100% used recursion in TREE. The results of the statistical test is that this is a very significant difference ($p < 0.0001$), and the effect size is 2.23 (large). This means that the null hypothesis was rejected, and we can conclude that a linear topology of the data structure definition is a significant factor.

However, we also have other experimental results on trees in which the use of recursion was much lower: problems SEARCH and MAX. These can be explained by distinguishing between the topology of the structure and the unfolding of the processing. Specifically, we suggest in RQ3 that the linearity

alluded to above may be inherent in the processing even if the underlying structure is not linear. In SEARCH the linearity comes about because we trace a single specific path in the tree. In MAX this effect is even stronger, because we know in advance exactly which path we are going to trace: to find the maximal element we always continue to the right son (RQ4).

Formally, we support these suggestions with two statistical tests. In the first test we compare the TREE and SEARCH problems. The topology of the data structure is the same: both are binary trees. So the independent variable is not the topology but the linearity of the processing — in SEARCH we have a single linear path in the tree, while in TREE we backtrack to explore different branches. The dependent variable is as usual the percentage of the participants who use recursion. The null hypothesis is that expected value of the percentage of the participants who use recursion is equal in both cases. The results were that 100% used recursion in TREE, and 67.6% did so in SEARCH. The results of the statistical test is that this is highly significant, with a p-value = 0.001178 and effect size of 1.2 (large). Thus the null hypothesis was rejected.

In the second statistical test the independent variable is again categorical with two levels, whether we know the direction of the path in advance or not. The dependent variable remains the method of processing (recursive or iterative), and the null hypothesis is that the expected value of the percentage of the participants who use recursion is equal in both cases. The alternative hypothesis is that the expected values of these percentages are different. The results were that in SEARCH 67.6% used recursion, and in MAX 34.2% did so. The results of the statistical test is that this is also significant, with a p-value = 0.009411 and effect size of 0.68 (medium). Thus the null hypothesis was rejected.

To summarize, our results indicate that the most important factor affecting the choice of processing style is *the topology of the computation path*. The more linear and obvious the path, the higher the tendency to use iteration. This has important implications regarding how we interpret the results. For example, we observed a strong tendency to process linked lists iteratively. But we suggest that this is not because of the list's linear structure, but rather because of the liner path of the processing. The linear structure of a linked list is just one situation in which such a linear path arises.

Another important factor is being input-oblivious, in the sense that the direction (and perhaps also the length) of the path is predetermined. For example, this is the reason that MAX is more iterative than SEARCH, and nearly as iterative as LIST (and indeed the statistical test showed that the difference between MAX and LIST is not significant, with a p-value = 0.110). This also indicates that emphasizing the underlying structure is the wrong consideration to apply, because the concrete path used in each specific instance is more important.

Supportive Evidence

In parallel to the execution of our experiment we became aware of an exercise in the data structures course in a university unrelated to us, which was very similar. The students in that course were required to merge two sorted linked lists into a single linked list. This is similar to our LIST problem. There were 284 students who submitted the exercise, so the sample was much larger than in our experiment. We requested and received a statistical summary of the approach they used in their solutions. The results were that 16.2% used recursion, very close to the result of 19.5% in LIST (Figure 2). Interestingly, the solution provided by the teaching staff was recursive, and the TA told us she thought this was the natural approach in this case. But obviously the students thought otherwise.

Factorial

In addition to previous problems which were described in Figure 1, our experiment also included the FACTORIAL question. This question is different from the other questions because it does not deal with a data structure. However, there is a similarity to LIST in that it is a linear structure that is defined recursively. We then applied a statistical test to see whether the fraction of participants using recursion was above 50%. 31.25% gave a recursive solution (Figure 3), with a significance p-value = 0.02591.

This result is interesting because factorial — together with Fibonacci — is a canonical problem used to

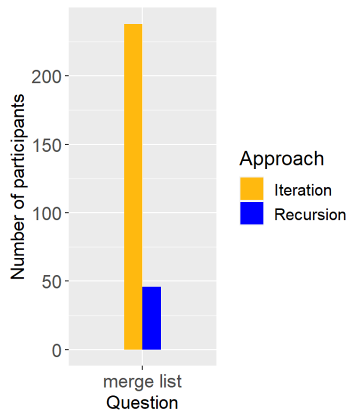


Figure 2 – Use of iteration and recursion in a data structures exercise.

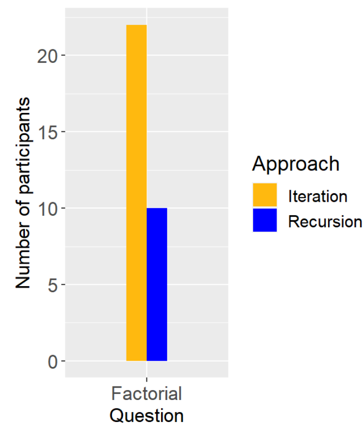


Figure 3 – Use of iteration and recursion in FACTORIAL.

teach recursion. This context may explain the higher use of recursion relative to LIST. But still, the vast majority did not use recursion.

4. Experiment 2: Comprehension

The second experiment complements the first experiment and examines the comprehension of iterative and recursive codes.

4.1. Research Questions

This experiment is limited to linear recursion, in the sense that only one recursive call is made each time. But we do want to distinguish between two types of such recursive calls. The first is tail recursion, in which the recursive call is the last statement that is executed by the function. It can therefore be followed in a similar way to iteration. The other is "classic" recursion, where some processing occurs after the recursive call. This may require a more general mental model, and is expected to be harder to follow. The research questions are:

RQ6. Is there a difference between the understanding of tail recursion and iteration?

RQ7. Is there a difference between the understanding of classic recursion and iteration?

RQ8. Can the code length, for example when recursive code is shorter, affect the relation of understanding recursive vs. iterative code?

4.2. Methodology

Our methodology is again to conduct a controlled experiment, this time to see which approach (iterative or recursive) programmers find easier to understand.

4.2.1. Experimental Materials

In this experiment we need problems that are not obviously iterative like traversing an array, and not obviously recursive like traversing a whole tree. All the problems are based on recursively-defined data structures, and linear processing. We saw in the first experiment that such problems are not necessarily natural to process using recursion, but perhaps it is easier to understand their recursive implementation?

The problems for the comprehension task are different from each other and have been chosen so that the functionality is realistic and not artificial. This is important because participants may waste time trying to understand what exactly the code is used for, when in fact it has no real use. Also, using contrived problems may increase the fraction of inaccurate and incomplete answers, leading to much variation in the content and length of the answers. All this is in contrast to a writing experiment, where made-up

Table 2 – Summary of comprehension problems used in the second experiment.

Name	*Data structure*	*Description*	*RQ*
LSEARCH	linked list	find a value	RQ6
MAX	binary search tree	find maximal value	RQ6
MERGE	linked list	merge two sorted lists	RQ6 RQ8
REVERSE	linked list	reverse a list	RQ7

problems like counting odd elements are useful. Ideally, it is desirable that both versions (recursive and iterative) have the same length, but this is not always possible.

The programming problems we used in this experiment are outlined in Table 2. The considerations for choosing these problems are as follows.

1. LSEARCH: Check if a certain value exists in a linked list. This problem was used by Benander et al. in their groundbreaking work comparing recursion and iteration, where they found that recursion is easier to understand (Benander et al., 1996). We therefore wanted to include a replication of their work in our experiment.

2. MAX: Find the maximal value in a binary search tree. This is the same problem as in the first experiment, but in this case we are interested in understanding the code. It is included as a representative of using a non-linear data structure.

3. MERGE: Given two sorted lists, merge them into one sorted list. There are several reasons for choosing this problem. First, it is an example of common cases where the recursive code is shorter. Also, it is an interesting case since the linear processing is over two lists at the same time.

4. REVERSE: Return a linked list in reverse order. This is interesting because during the processing we make a change to the structure itself. Importantly, the change is done after the recursive call, so this is a classic and not tail recursion.

Note that in the previous experiment the main difference between problems was in the underlying data structures, and the differences in processing were rather minor. Here, on the other hand, we focus on processing the same structure. The first problem, LSEARCH, is basically just a traversal of a linked list, similar in spirit to the problems in experiment 1. The second problem, MAX, represents the alternative platform of a search tree instead of a linked list. The last two problems involve more complex processing, which may lead to more complex code. In MERGE we traverse two lists in parallel, and in REVERSE we modify the list as we traverse it.

4.2.2. Experiment Execution

This experiment was also conducted online using the Qualtrics platform. We described the experiment as an experiment on code comprehension of recursive and iterative code.

Each participant was asked about all four problems. Initially we reminded them about binary search trees and asked them to understand the code of the MAX problem. Then we gave a short introduction on linked lists, and presented them with SEARCH, REVERSE, and MERGE. For each problem, one version — either recursive or iterative — was chosen at random. Participants never got both versions of the same problem.

In recruiting participants we again emphasized recruiting participants from industry. This has two reasons. One is to be more relevant to the "real life" of software development. In addition, we wanted participants who were farther removed from the programming patterns common to academic studies. Participants were not compensated for their participation.

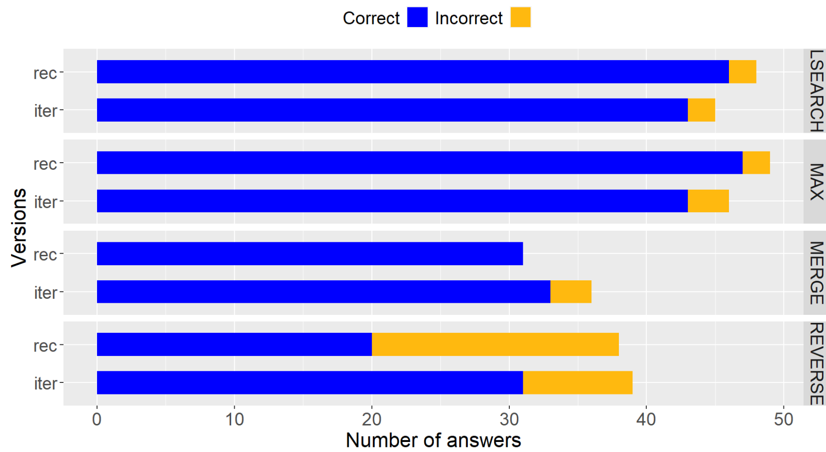


Figure 4 – Correctness results.

4.3. Results

4.3.1. Participants and demographic background

The experiment was conducted from August to November 2022. All told, we had 101 participants who answered at least one problem. 70% of the respondents reported their gender: 75% of them were men and 25% women. In terms of education, 73% of the respondents reported their education: most of the participants had an academic background, but not all of them. 40 had a BSc degree or were studying for one, and 26 had an MSc or were MSc students. 8 were PhD students. We also asked about professional programming experience (excluding studies). 71% of them reported it. 22 participants had experience between 0–2 years, 50 had more than 3 years of experience, and the median is 3.5.

4.3.2. Correctness Results

We measured two response variables in the experiment: the correctness of the answer and the time it took for the participants.

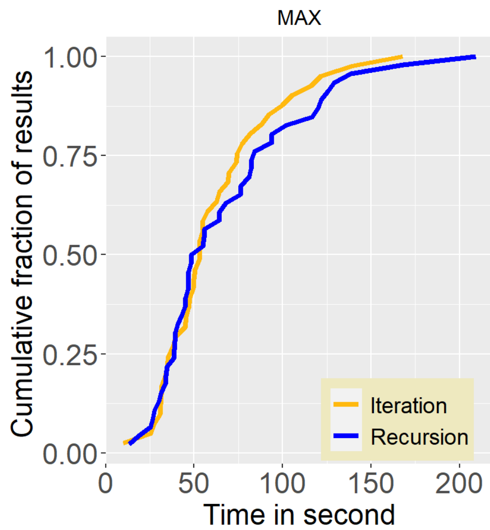
In judging correctness we want to emphasize the essence of the code. In other words we want to avoid penalizing participants for superficial mistakes, provided they did understand the basic functionality.

For example, the correct answer to the MAX problem is that it finds the maximal element in the search tree, which is also the rightmost element. A possible mistake is saying that this is the minimum element. But a likely interpretation for this mistake is that the programmer actually understood the code itself, and was just confused about which son is larger and which smaller — information that does not appear directly in the code. We therefore accepted his answer even though it is technically incorrect.

On the other hand, in the REVERSE question, a common answer was that the function returns the last element of the list. This answer is factually true: the function returns a pointer to the last element of the input list. But this answer misses the essential part of the code which reverses the list, and makes the last element first. Therefore we did not accept this answer even though technically it describes a correct fact.

Figure 4 presents the number of correct answers out of all the answers for each version of each question. We conducted a statistical test in each of the questions see whether there is statistically significant difference between the versions. The independent variable is the nature of the processing (recursion or iteration) and the dependent variable is the fraction of correct answers. The dependent variable is binomially distributed, so we used the Z-test.

It is easy to see that in the first 3 questions there is little difference between the versions, and the statistical test confirms that this is not statistically significant. So from the perspective of correctness, the answers to RQ6 and RQ8 is that there is no difference between iteration and recursion. The only problem where there is a statistical difference is REVERSE: in the iterative version of REVERSE, 31 of 39 gave a correct



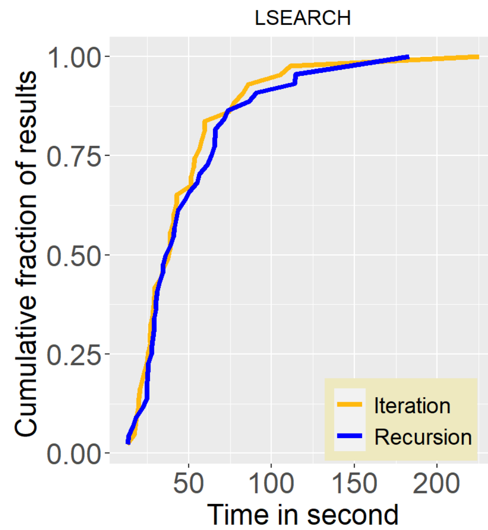


Figure 5 – Cumulative Distribution Functions (CDFs) of time to correct solution for MAX.

Figure 6 – CDFs of time to correct solution for LSEARCH.

answer. But in the recursive version only 20 gave a correct answer out of 38. The statistical test gave p-value = 0.0244 and effect-size 0.578 (Medium). This shows that for RQ7 there is a difference. We will discuss this after we analyze the time differences.

4.3.3. Time Results

Our experiments check the time for developers to understand each of the codes. We would like to check whether the expected times in different versions are equal. For this we will use a t-test between subjects, where the null hypothesis is that they are equal, and the alternative hypothesis is that the expected values are different. Only correct answers were considered in this analysis.

The difference between the recursive version and the iterative version in the MAX question is not significant, since the p-value = 0.4808, and Fig. 5 shows that indeed the distributions are very similar to each other. This finding, together with the statistical result for the percentage of correct answers, shows that there is no essential difference in terms of code comprehension as far as this question is concerned.

Similarly, for the question LSEARCH we got that p-value = 0.6802, that is, the differences are not statistically significant, and Fig. 6 shows that again the distributions are similar. The combination of statistical tests again indicates that there are no differences in the difficulty of understanding the two versions. This is contrary to the results of Benander et al., who received a statistically significant result in the LSEARCH problem (Benander et al., 1996), but consistent with the replication results of (McCauley et al., 2015). In our view, a possible explanation for the difference between the results is the difference in program lengths. In Benander et al.'s experiment the recursive version came out more readable, but this could be because its code was shorter. Another possible reason for the difference may be that we appealed to experienced programmers, while Benander et al. used students from a first data structures course. This is meaningful because recursion is a major topic in such a course, but is not very common in everyday work. So the students may have been more "into" recursion than the professional programmers.

Taken together it can be seen that for simple linear processing, where the code segments are approximately the same length, using iteration or recursion does not dramatically affect the code comprehension. This answers research question RQ6.

In MERGE we can see in Figure 7 that the distributions are separated, and understanding the iterative version takes a longer time (the CDF is shifted to the right). This is statistically significant with p-value = 0.02434 and effect-size (Cohen's d) of 0.581 (medium). While this problem is a classic linear

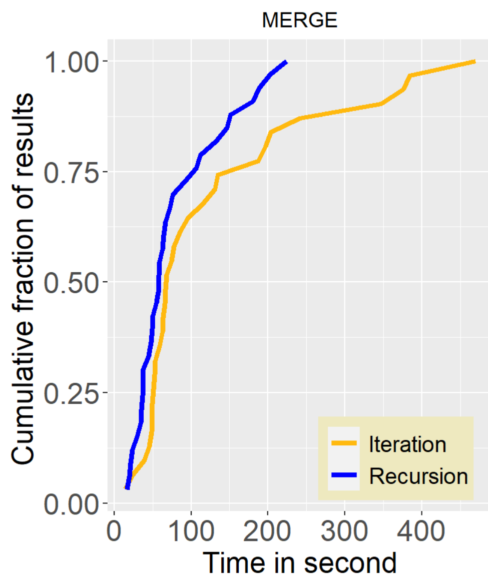


Figure 7 – Cumulative Distribution Functions (CDFs) of time to correct solution for MERGE.

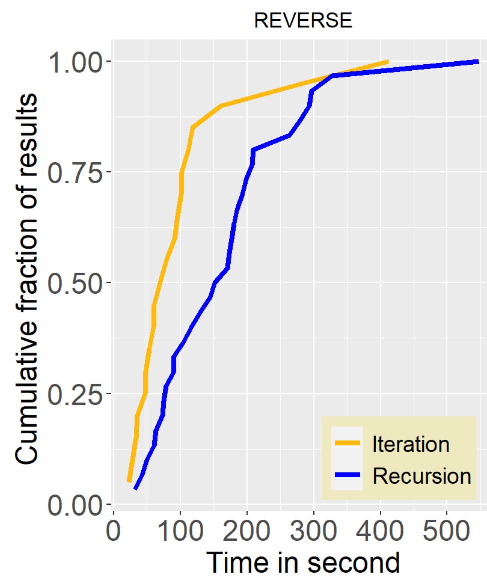


Figure 8 – CDFs of time to correct solution for REVERSE.

processing of lists, like the previous two question, in this one the recursive code is significantly shorter than the iterative alternative. This is similar to the above-quoted result by Benander et al., that the recursive code is more readable since it is shorter, and answers RQ8.

Finally, we turn to the last problem REVERSE. We can see in Figure 8 that the distributions are separated, but in contrast to the previous result, in this case understanding the recursive version takes a much longer time (the CDF is shifted to the right). The results are very significant with p-value = 0.002819 and the effect size (Cohen's d) is 0.95 (large). This answers RQ7.

In order to analyze the results of REVERSE we begin with the code. The recursive version is as follows:

```
1 def func(node):
2     if (node == None):
3         return node
4     if (node.next == None):
5         return node
6     node1 = func(node.next)
7     node.next.next = node
8     node.next = None
9     return node1
```

Our interpretation is that REVERSE is an example of a recursive function with non-trivial commands that have to be executed after the recursive call (on the "passive flow", lines 7-8). But when you read such a function and reach the recursive call, you do not know this yet. Therefore you cannot really know what the recursive call does, which makes the recursion harder to trace. This analysis is consistent with (Scholtz & Sanders, 2010; Sanders et al., 2006; Götschi, Sanders, & Galpin, 2003) and others, who claim that recursion of the above type requires a more accurate "mental model". The alternative iterative version is more transparent, due to the fact that we can see the mechanism change each element one after the other. This is likely to be easier to follow.

This analysis is bolstered by the nature of the errors that participants made when trying to understand the function. The most common error (7 from 18) was to state that it returns the last element in the

list. We speculate that they reached this conclusion by tracing the code in the following way. First, they noted that the function returns node1 (in line 9). Then they saw that node1 is assigned its value from the recursive call (in line 6). As there are no other assignments to node1, this implies that its value percolates back through all the recursive calls. So they looked for the termination condition, and found that the last element (the one with next == None) is returned (lines 4-5). But in this tracing they skipped the two hard-to-understand lines that change the structure and cause the list to be reversed after the exit from the recursion (lines 7-8). This analysis is consistent with previous studies that found that a common misconception is that participants tend to think that a recursive method stops execution when the base case is reached (Dicheva & Close, 1996; Segal, 1995).

Additional work is needed to verify all this. We are planning followup experiments using other recursive versions, and specifically using tail recursion.

5. Threats to Validity

Construct validity Our experiment measures understanding according to the criteria of measuring times and checking the correctness of the answer. These are the commonly used metrics (Rajlich & Cowan, 1997). But time and correctness are only proxies for understanding. Time can be affected by many external factors that are not relevant to the understanding itself. Also, it can be that a correct description does not derive from a deep and thorough understanding of the code, but from an informed guess, prior knowledge, and more.

Internal validity Our conclusions may not follow from the experiments for several reasons. In comprehension experiments it is possible that the observed differences are not a result of the intended difference in the independent variable, namely iteration vs recursion, but maybe something else. For example, the names of variables in the iterative and recursive codes of the REVERSE problem are different. Due to the nature of the iterative code, there are more variables in it, and their names are slightly more informative, such as current and prev. Of course, these are still names that do not provide information about the functionality of the code, but they help a little in tracing it. However, this is probably not really a problem in this case because the differences were very significant in both time and correctness, so it is unlikely that this difference affects the results themselves.

External validity Our experiment included only a few pieces of code, and although we tried to reach a diverse audience in terms of programming experience, gender, and education, it is still likely that it is not fully representative. The study should be replicated to ensure its validity and map out its generalizability.

6. Conclusions

Our research consisted of two stages, concerning code development (experiment 1) and code comprehension (experiment 2). In the first experiment we identified several factors affecting the use of iteration vs. recursion. The first is the *topology* of the data structure, and specifically whether it is linear. The second is the trajectory of the *processing*, and again whether it is linear. The third is whether this is *oblivious* or input-dependent. In retrospect, it appears that all three factors reflect the degree to which the problem is amenable to a straightforward serial solution, or in other words, to a simple iterative solution. This is reflected in the results, where the more linear and oblivious problems were more often solved using iteration.

In the second experiment we observed an apparent contradiction between MERGE and REVERSE: in MERGE the recursive version was easier to understand, while in REVERSE it was dramatically harder. We believe that this difference indicates that a more nuanced classification is needed: the distinction should not be only between iteration and recursion, but also between different types of recursion. Specifically, MERGE (and also LSEARCH and MAX) are solved using a tail recursion which is similar to iteration: the processing is essentially done one element at a time, from beginning to end. But REVERSE, in contradistinction, requires a more classic form of recursion, in which part of the processing occurs

on the exit path, after the recursive call. So each element is visited twice, in the call and in the return, and in the return it is not just passed back passively, but manipulated in a non-trivial manner. This is what makes the recursive version more complex. And in such a case, if there is a simple iterative alternative where the transition is more linear and serial, then it is likely that the iterative code will be more understandable.

Taken together, these two experiments suggest possible cognitive underpinnings of the choice between recursion and iteration. Algorithmic thinking is inherently serial—we define algorithms as a sequence of steps, and find it hard to think about things that happen in parallel. Likewise, it is easier to think about linear processing, where we handle elements one after the other, and do not need to return to those we have already visited (less linear). Such thinking is analogous to using iteration. Recursion is useful when such thinking is insufficient: when we need to save state for use in additional processing later (as in REVERSE), or when the basic structure is not linear (as in TREE).

The above observations have several implications for programming practice. One is to beware of dogmas. For example, it is often thought that using a recursive definition implies a preference for recursive processing. We believe that a recursive definition may be a necessary condition for using recursive processing, but it is not a sufficient condition. Other considerations also need to be applied.

7. Experimental Materials

The experimental materials are available in:
https://doi.org/10.5281/zenodo.8266266

8. Acknowledgements

This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 832/18).

9. References

Aqeel, A., Peitek, N., Apel, S., Mucke, J., & Siegmund, J. (2021). Understanding comprehension of iterative and recursive programs with remote eye tracking. In *32nd Workshop Psychology of Programming Interest Group.* Retrieved from https://ppig.org/papers/2021-ppig-32nd-aqeel/

Benander, A. C., Benander, B. A., & Pu, H. (1996). Recursion vs. iteration: An empirical study of comprehension. *J. Syst. Softw.*, *32*(1), 73–82. doi: 10.1016/0164-1212(95)00043-7

Bloch, S. (2003, May). Teaching linked lists and recursion without conditionals or null. *J. Comput. Sci. Coll.*, *18*(5), 96-108.

Choi, S. G. (2021). *United States Naval Academy, introduction to computer science.* https://www.usna.edu/Users/cs/choi/si204/lec/l36/lec.html. (Accessed: 2022-01-16)

Dicheva, D., & Close, J. (1996). Mental models of recursion. *J. Educational Computing Research*, *14*(1), 1-23. doi: 10.2190/AGG9-A5UD-DEK0-80EN

Elkner, J., Downey, A. B., & Meyers, C. (2012). *How to think like a computer scientist: Learning with python.* https://www.cs.swarthmore.edu/courses/cs21book/build/ch18.html. (Accessed: 2022-01-16)

George, C. E. (2000). EROSI - visualising recursion and discovering new errors. In L. B. Cassel, N. B. Dale, H. M. Walker, & S. M. Haller (Eds.), *Proc. 31st SIGCSE Technical Symp. Computer Science Education* (pp. 305–309). Retrieved from https://doi.org/10.1145/330908.331875 doi: 10.1145/330908.331875

Götschi, T., Sanders, I. D., & Galpin, V. (2003). Mental models of recursion. In S. Grissom, D. Knox, D. T. Joyce, & W. P. Dann (Eds.), *Proc. 34th SIGCSE technical symp. Computer science education* (pp. 346–350). doi: 10.1145/611892.612004

Greer, J. E. (1989). A comparison of instructional treatments for introducing recursion. *Computer Science Education*, *1*(2), 111–128. doi: 10.1080/0899340890010204

Haberman, B., & Averbuch, H. (2002). The case of base cases: Why are they so difficult to rec-

ognize? *Proc. 7th conf. Innovation & Tech. in Comput. Sci. Education*, 84-88. doi: 10.1145/637610.544441

Kahney, H. (1983). What do novice programmers know about recursion. *Proc. SIGCHI Conf. Human Factors in Computing Systems*, 235–239. doi: 10.1145/800045.801618

Kessler, C. M., & Anderson, J. R. (1986). Learning flow of control: Recursive and iterative procedures. *Human-Computer Interaction*, *2*(2), 135–166. doi: 10.1207/s15327051hci0202_2

Kiesler, N. (2022). Mental models of recursion: A secondary analysis of novice learners' steps and errors in Java exercises. In S. Holland, M. Petre, L. Church, & M. Marasoiu (Eds.), *33rd Workshop Psychology of Programming Interest Group* (pp. 226–240). Retrieved from `https://ppig.org/papers/2022-ppig-33rd-kiesler/`

Lee, E., V., S., Beth, B., & Lin, C. (2014, July). A structured approach to teaching recursion using cargo-bot. In *10th Conf. International Computing Education Research* (p. 59–66). doi: 10.1145/2632320.2632356

McCauley, R. A., Hanks, B., Fitzgerald, S., & Murphy, L. (2015). Recursion vs. iteration: An empirical study of comprehension revisited. In *46th ACM Tech. Symp. Comput. Sci. Education* (pp. 350–355). Retrieved from `https://doi.org/10.1145/2676723.2677227`

Mirolo, C. (2012, Sept). Is iteration really easier to learn than recursion for CS1 students? In *Proc. 9th Intl. Conf. Comput. Education Research* (p. 99-104). doi: 10.1145/2361276.2361296

Pattis, R. E. (2006). *Recursion on linked lists, advanced programming/practicum.* `https://www.cs.cmu.edu/~pattis/15-1XX/15-200/lectures/llrecursion/index.html`. (Accessed: 2022-01-23)

Rajlich, V., & Cowan, G. S. (1997, Mar). Towards standard for experiments in program comprehension. In *5th International Workshop on Program Comprehension* (pp. 160–161). doi: 10.1109/WPC.1997.601284

Sanders, I. D., Galpin, V., & Götschi, T. (2006). Mental models of recursion revisited. In R. Davoli, M. Goldweber, & P. Salomoni (Eds.), *Innovation & Tech. in Comput. Sci. Education* (pp. 138–142). doi: 10.1145/1140124.1140162

Scholtz, T. L., & Sanders, I. D. (2010). Mental models of recursion: investigating students' understanding of recursion. In R. Ayfer, J. Impagliazzo, & C. Laxer (Eds.), *Innovation & Tech. in Comput. Sci. Education* (pp. 103–107). doi: 10.1145/1822090.1822120

Segal, J. (1995). Empirical studies of functional programming learners evaluating recursive functions. *Instructional Science*, *22*, 385–411.

Snyder, W. (n.d.). *Recursion and linked lists.* `https://www.cs.bu.edu/fac/snyder/cs112/CourseMaterials/LinkedListNotes.Recursion.html`.

Wiedenbeck, S. (1989, January). Learning iteration and recursion from examples. *International Journal of Man-Machine Studies*, *30*(1), 1–22. doi: 10.1016/S0020-7373(89)80018-5

Wirth, N. (1976). *Algorithms + data structures = programs*. Englewood Cliffs, N.J.: Prentice-Hall.