Understanding Comprehension of Iterative and Recursive Programs with Remote Eye Tracking

Arooba, Aqeel
Chemnitz University of Technology

Norman Peitek
Leibniz Institute for Neurobiology

Sven Apel
Saarland University
Saarland Informatics Campus

Jonas Mucke
Chemnitz University of Technology

Janet Siegmund
Chemnitz University of Technology

Abstract

Background: There have been many studies on the teaching and learning of programming styles, including *recursion* and *iteration*. Programming styles have been studied from the point of view of mental and conceptual models, program comprehension, common misconceptions, and how to teach them effectively. Recent studies suggest that students tend to understand iterative programs more efficiently than recursive programs. However, there is little empirical evidence as to why this might be the case.

Objective: To create better teaching practices for students, we first have to understand how students understand recursion and iteration. We aim to investigate this phenomenon and identify factors that might drive the understanding of iterative and recursive programs by students.

Method: We conducted a remote eye-tracking study to record students' visual attention as they were comprehending simple iterative and recursive programs. A total of 117 students participated in the study, and we collected behavioral and visual-attention data.

Results: We found no clear indication that programming style affects how well and fast students understand iterative and recursive programs. Regarding visual attention, we observed that students follow a comparable reading behavior for both iterative and recursive programs. However, we found different reading behaviors when comparing students who correctly understood programs with students who did not.

Conclusion: It can be said that for students, there is no difference in efficiency in understanding iterative and recursive algorithms. The visual attention of that same group is equally indistinguishable between top-down and bottom-up comprehension. But it seems that the different programming styles have a different approach to understanding the source code, so other beacons in the code become more relevant and these are visited more often in the course of the comprehension process.

Future Work: To derive more general conclusions, further studies are necessary, for example, with more advanced code (e.g., with mutual recursion) or letting students write code themselves. Measuring the cognitive load (e.g., with pupil dilation) could also provide more rigorous insights into the underlying cognitive process of program comprehension.

1. Introduction

There are many ways to teach programming.Some say it is important to teach computational thinking (Lye & Koh, 2014), others say making a connection to real-life examples is critical (Barjaktarovic, 2012). For example, to teach recursion, the Russian Matryoshka doll can be used, where each hollow wooden doll contains a similar but smaller version of itself (Wu, Dale, & Bethel, 1998). Another approach to introduce recursion is by a variety of applied examples of mathematical functions, such as computing the factorial, the Fibonacci sequence, or Euclids algorithm (Benander, Benander, & Pu, 1996). Connecting to mathematical skills that students have acquired in high school builds on already developed problem-solving skills, which may be translated intuitively into source code (Winslow, 1996).

Iteration and recursion are core problem-solving techniques and an important component in computer science. The underlying concept of both is to execute a set of instructions repeatedly. The difference

```java
public static void main(String args[]) {
    int num = 3;
    System.out.println(findfibonacci(num));
}

static int findfibonacci(int n) {
    int a = 0, b = 1, c;
    if (n == 0)
        return a;
    for (int i = 2; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

(a)

```java
public static void main(String args[]) {
    int num = 3;
    System.out.println(findfibonacci(num));
}

static int findfibonacci(int n) {
    if (n <= 1)
        return n;
    return findfibonacci(n - 1) + findfibonacci(n - 2);
}
```

(b)

Figure 1 – Java snippets to compute the Fibonacci number in an (a) iterative implementation and (b) recursive implementation.

between them is that recursion is simply a function call, in which the function being called is the same as the one making the call, whereas iteration is when a set of statements inside a loop is repeatedly executed until a certain termination condition is met.

Iteration and recursion require different ways of thinking, which may make one or the other approach more intuitive to start with. Specifically, iteration requires forward reasoning, and recursion requires backward reasoning. In *forward reasoning*, one starts with the data for an initial state and works toward a goal. For example, to compute the Fibonacci number of 4 (cf. Figure 1(a)), a forward-reasoning approach could be:

1. Get the first number (i.e., 0)
2. Get the second number (i.e., 1)
3. Calculate the third number by adding the first and second (i.e., $0+1=1$)
4. Calculate the fourth number using second and third (i.e., $1+1=2$)

In *backward reasoning*, the data for an initial state need to be found to solve a problem. Here, problems are divided into sub-problems; if the solution for the sub-problem is not already calculated, we need to calculate it on the way. For example, computing the Fibonacci number of 4 with a backward-reasoning approach (cf. Figure 1(b)) would work as follows:

1. `Fibonacci(4)` $\rightarrow$ go and compute `Fibonacci(3)` and `Fibonacci(2)`.
2. `Fibonacci(3)` $\rightarrow$ go and compute `Fibonacci(2)` and `Fibonacci(1)`.
3. `Fibonacci(2)` $\rightarrow$ go and compute `Fibonacci(1)` and `Fibonacci(0)`.
4. `Fibonacci(1)` will return 1 and `Fibonacci(0)` will return 0.

Ginat suggested students find recursion difficult, because it requires backward reasoning, which is unintuitive for students (Ginat, 2005). He assumes most of the problems students have solved before encountering recursion require forward reasoning working from the initial state to the goal state and thus recursion requires a new way of thinking. Since students have not been taught to use backward reasoning, they struggle with recursion.

There is empirical evidence for both, that students either prefer iteration or recursion. For example, Gal-Ezer and Harel suggest recursion is "one of the most universally difficult concepts to teach" (Gal-Ezer & Harel, 1998). Similarly, Roberts states that students perceive recursion as "obscure, difficult and mystical" when they are first introduced to it (Roberts, 1986). Benander and others found that recursive search and copy routines were no more difficult to comprehend than iterative routines (Benander et al., 1996). Turbak has observed an improvement of students performance in exams after changing the course structure and introducing recursion first as a problem-solving strategy. Iteration is then presented as a particular pattern of recursion (i.e., tail recursion). Finally, loop constructs are presented as concise

idioms for iterative patterns (Turbak, Royden, Stephan, & Herbst, 2001).

A common viewpoint is that iteration is easier to comprehend for students. Kessler and Anderson investigated the relationship between writing recursive code and writing iterative code. The participants in their study were undergraduate students with little or no programming experience. The study was conducted over two sessions in two days, using a language designed specifically for the study. They found positive transfer from writing iterative functions to writing recursive functions, but not vice versa (Kessler & Anderson, 1986).

McCauley and others (Mccauley, Grissom, Fitzgerald, & Murphy, 2015) replicated the study of Benander and others (Benander et al., 1996). McCauley and others observed that students found it significantly more challenging to comprehend recursive code than iterative code.

In a study on how middle-school students learn recursion, Anzai and Uesato found that learning iteration first facilitated the learning of recursion. The participants in the study did no programming and did not read code, but instead saw a specification on how to compute a solution (iteratively or recursively) for some instance of a problem and should calculate the solution to a larger instance (Anzai & Uesato, 1982).

Iteration (or looping) is a common phenomenon in our everyday lives. The availability of real-world analogs, therefore, facilitates the development of a mental model of control flow in iteration. A good example of this scenario would be a music player, which keeps playing music unless the user presses "stop" or it is the end of the playlist. By contrast, for recursion, often artificial examples with unclear real-world analogies are used. Popular examples are computing Fibonacci numbers and Euclid's algorithm, which can concisely illustrate recursive computations. However, as they may not succeed in motivating the need for recursion in a broader setting, students may perceive that recursion is only used for these mathematical functions (Michaelson, 2015; Shriram, 2020).

To teach recursion to students, Felleisen and others suggest doing it naturally (Felleisen, Findler, Flatt, & Krishnamurthi, 2018). Specifically, they argue that recursion arises from self-references in data, so recursive data suggests recursive solutions. Felleisen and others teach a design "recipe", in which students describe their programs (or functions) data structures, and identify self-references in these data. Next, they design a "template" that explains how the structure of data explains the structure of the solution. This function is structurally recursive, that is, a function that consumes structured data, typically decompose their arguments into their immediate structural components and then process these components. If one of the immediate components belongs to the same class of data as the input, the function is recursive. For this reason, these functions are called structurally recursive functions (Felleisen et al., 2018). While structurally-designed functions make up the vast majority of code in the world, some problems cannot be solved with a structural approach to design. To solve such complicated problems, programmers use generative recursion, a form of recursive algorithms that generate an entirely new piece of data from the given data and recur on it (Felleisen et al., 2018). We focus on structural recursion in this study, because it occurs most frequently.

Despite numerous studies on the dichotomy of recursion and iteration, there is still a lack of understanding in the students' underlying ways of thinking (Rinderknecht, 2014; Mccauley et al., 2015). Due to this gap in understanding, it is unsettled how to ideally teach recursion and iteration.

One important aspect of program comprehension is observing the way programmers read source code. Eye tracking has proved useful to observe programmers reading source code and answer such fundamental research questions on program comprehension (e.g., (Busjahn et al., 2015), (Peitek, Siegmund, & Apel, 2020)). Eye-tracking usually requires direct measurement taken from the eye in one of several ways, but principally there are three categories: (a) measurement of the movement of an object (i.e., special contact lens) attached to the eye, (b) optical tracking without direct contact to the eye (i.e., video-based eye trackers), or (c) measurement of electric potentials using electrodes placed around the eyes (i.e., electrooculography (EOG)). However, conducting a study with one of these techniques was

especially challenging in the current pandemic. To allow for safe and parallel observation, we used an inferential technique, using a tool called REYEKER. The original image is blurred to distort text regions and disable legibility, requiring participants to click on areas of interest to deblur them to make them readable. REYEKER collects each mouse click in terms of (x, y) coordinates. These serve as approximation of participants visual attention. While REYEKER naturally can only track visual attention to a limited degree as information from peripheral vision is greatly reduced, it allows researchers to get a basic understanding of developers' reading behavior as we believe that the results obtained are valid indications of the focus of attention. We hope to repeat these studies in the future using more direct techniques.

To gain more insights into how students understand iterative and recursive programs, we conducted a remote eye-tracking study. Specifically, we observed how 117 students of two introductory programming courses read 8 algorithms in iterative and recursive style.

In a nutshell, we found that students' behavior and visual attention is not affected by the programming style. In summary, we make the following contributions:

- We describe an experiment design that investigates the underlying ways of thinking when students understand iterative and recursive programs.
- We provide empirical evidence that, for early students, there is no significant difference between iterative and recursive style.
- We share a complete replication package, which includes all snippets, acquired data, and analysis scripts on the project's website.[1]

2. Experiment Design

2.1. Objective

With our study, we aim at shedding more light on how students approach understanding iterative and recursive algorithms. Specifically, we evaluate whether the implementation style affects students' behavior and visual attention. Furthermore, to be able to entangle the effect of implementation style from comprehension strategy, we evaluate whether meaningful vs. meaningless identifier names also affect students' behavior and visual attention. We state two research questions to clarify our objective:

RQ1: Do implementation style and comprehension strategy affect students' response times and correctness when understanding source code?

RQ2: Do implementation style and comprehension strategy affect students' visual attention when understanding source code?

As prior studies point in both directions (in favor of recursion or iteration), we formulate questions and not hypotheses. Note that we include the comprehension strategy as an independent variable, because our pilot studies, which we have conducted to select suitable snippets and tasks, suggested that it may affect how students understand iterative and recursive snippets. With comprehension strategy, we refer to the classic dichotomy of top-down and bottom-up comprehension, each having a different cognitive approach to comprehend a snippet. *Top-down comprehension* describes that programmers quickly form a hypothesis about a program's purpose and step by step refine this hypothesis by looking into the source code in more detail (Brooks, 1983). *Bottom-up comprehension* describes that programmers start comprehending source code by looking at individual statements and step by step integrate these statements to semantic chunks until having reached a high-level understanding of source code (Pennington, 1987). Just like recursion and iteration require backward and forward reasoning, top-down and bottom-up comprehension require different reasoning strategies. Thus, an interaction between the two (backward/forward reasoning and top-down/bottom-up comprehension) may very well exist, which we might be able to detect when including both as factors. This might also shed some light on why either recursion or iteration may be more accessible.

[1] `https://github.com/brains-on-code/IterationVsRecursion`

2.2. Independent Variables

Our study design contains two independent variables:

- Programming style (i.e., iterative vs. recursive programs).

  Iteration is a set of statements that are repeatedly executed a specified number of times or until a condition is met. The classic mechanism of expressing iterations in programming code is through loops. Loops can be categorized as counter controlled loops (i.e., `for` loops) or condition controlled loops (i.e., `while` and `do while` loops) (Tsaramirsis, Al-Jammoor, & Buhari, 2014). In this study, we explore both counter and conditioned-controlled loops.

  On the other hand, recursion is a technique of solving a problem where the solution depends on solutions to smaller instances of the same problem. This can be done by functions, subroutines procedures, subroutines, functions, or algorithms that call themselves from within their own code (Sulov, 2016). Apart from classifying the recursive functions based on the input data and how they process it, recursion is categorized according to the number and type of the recursive function calls. A common classification distinguishes the types linear (of which tail recursion is a special case), multiple, nested, and mutual recursion (Rubio-Sánchez, Urquiza-Fuentes, & Pareja-Flores, 2008). In this study, we explore structural and generative recursion by examples from different categories, including linear recursion, tail recursion, and multiple recursion. The examples for nested and mutual recursions were not included in the study, as they did not meet the snippet selection criteria discussed in Section 2.4.

- Comprehension strategy (i.e., top-down vs. bottom-up comprehension).

  We operationalized the comprehension strategy by using meaningful versus meaningless identifier names in the source code snippets, which has been shown to elicit top-down or bottom-up comprehension strategy (Siegmund et al., 2017).

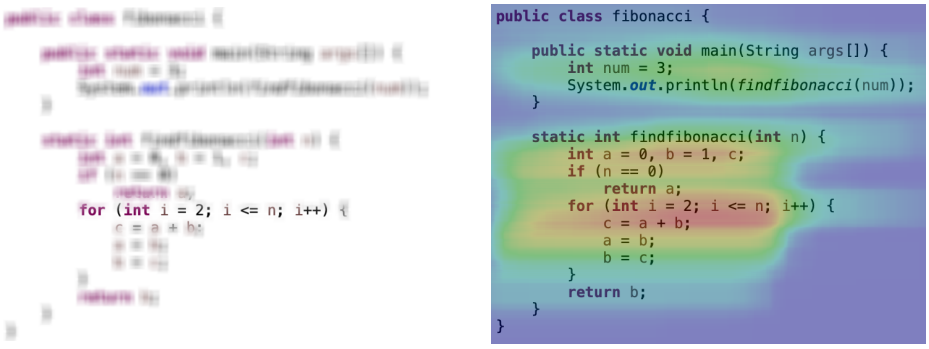
  For bottom-up comprehension, we need to ensure that participants go through the source code statement by statement. To this end, we use the same methodology used by Siegmund and others (Siegmund, Kastner, et al., 2014): We obscured identifier names, such that they did not convey the meaning of a variable and method, but only its usage.

  For top-down comprehension, we used meaningful identifiers that provide semantic cues to hint a programs purpose and set corresponding expectations (Siegmund et al., 2017).

2.3. Dependent Variables

As dependent variables, we analyze response time, correctness, and visual attention. We define response time as the time when participants first start viewing a snippet until the time they submit their answer. Correctness refers to whether students submit a correct answer to the task of what a source code would return (e.g., a sorted array after executing bubble sort).

To operationalize visual attention, we used the tool REYEKER to present source code to participants and to remotely track where participants are looking at in the source code (Mucke, Schwarzkopf, & Siegmund, 2021). To this end, REYEKER blurs a previously specified image of source code by applying a filter; only the part on which a participant clicks is readable. REYEKER collects each mouse click in terms of (x, y) coordinates. These serve as approximation of participants' visual attention. It logs the analysis of click data that displays points of focus for all source code snippets, this data will reveal if there is a difference in reading behavior when students encounter recursive or iterative programs. In Figure 2, we show an example of how participants view source code. The deblurred area in Figure 2 (a) almost entails one entire line of source code, which approximates where the participant is looking at. By clicking on another area, the current area will be blurred, and the clicked area will be deblurred. Figure 2 (b) shows a heatmap of the same code snippet. It illustrates the weighted average of all participants' data.



(a) (b)

Figure 2 – Figure (a) visualizes the user view of the ongoing evaluation study, using the following parameters: The selected shape is a rectangle with a width of 200 pixels and a height of 1 pixel. The transition range amounts to 30 pixels. The filter was set to 8 pixels in the x and y-axis. Figure (b) shows a heatmap of the same code snippet. It illustrates the weighted average of all participants' data.

2.4. Material

As material, we selected eight source code snippets. We summarize the snippets in Table 1. These snippets proved the best choice based on a number of pilot studies. In these pilot studies, we evaluated how long participants needed and how many snippets we could ask them to understand. We also asked the participants to estimate the difficulty of the snippets Notably, we found that meaningful identifiers affected how they worked with iterative and recursive styles, such that participants may take advantage of previous knowledge or experience when comprehending algorithms. For example, feedback on algorithms, such as finding the factorial of a number or reversing a string, suggested meaningful identifiers helped the participants to identify the purpose of the algorithm. Thus, we decided to include the comprehension strategy as a factor. For each algorithm, we created four versions for each snippet:

- Recursive Top-Down (R_TD)
- Recursive Bottom-Up (R_BU)
- Iterative Top-Down (I_TD)
- Iterative Bottom-Up (I_BU)

Snippet	Iterative		Recursive	
	LOC	Response Time [s]	LOC	Response Time [s]
BubbleSort	20	514 ± 54	19	826 ± 145
Factorial	15	504 ± 379	13	329 ± 71
Fibonacci	19	397 ± 218	13	526 ± 144
Reverse String	22	216 ± 89	14	208 ± 22
Integer to Binary	18	272 ± 86	14	219 ± 82
Binary Search	25	225 ± 65	25	331 ± 89
Prime Factors	24	420 ± 98	19	468 ± 323
Multiply Matrix	33	262 ± 42	25	381 ± 26

Table 1 – All snippets used in the study and the lines of code they contain and the mean response time during pilot studies.

2.5. Task

We asked participants to enter the result of the final print statement for all presented snippets. For example, for the Fibonacci snippets of Figure 1, the correct output is "2". The rationale of fixing the task to computing the output is that we aimed to eliminate the chance that the kind of task affects participants

Demographic Data	
Male	78
Female	25
Age	22.4 ± 5
Learning Programming (in years)	2.2 ± 1.9
Java Programming (in years)	0.5 ± 1.9
Professional Programming (in years)	0.4 ± 2.8
Recursion Understanding	2.9 ± 0.8
Iteration Understanding	3.2 ± 0.9
Skills in Comparison to Peers	2.8 ± 0.7

Table 2 – Demographic data of our participants. To measure experience, we used a validated questionnaire Siegmund, Kästner, et al. (2014). Participants estimate that they have a better understanding of iterative programs.

comprehension strategy (Dunsmore & Roper, 2000).

2.6. Participants

We recruited participants from Chemnitz University of Technology who were enrolled in undergraduate and graduate programs. Participants were compensated with a bonus point for their assignments they needed to complete to be admitted to the final exam. All participants had a fundamental understanding of Java and object-oriented programming (i.e., passed, at least, an introductory programming class). In Table 2, we provide an overview of our participants' demographics and programming experience.

2.7. Experiment Procedure

We used a randomized within-subjects design, which we visualized in Figure 3: Each participant understood snippets in all four versions. The order of conditions was randomized, and no participant saw the same algorithm twice. After the four snippets, participants could optionally work with four more snippets, again in all four versions in a randomized order with no repeating algorithms (57 participants used that option).

Figure 3 – Visualization of our experiment design. The order of the comprehension conditions was randomized.

Before the actual experiment, participants watched a video to explain the experiment procedure. Then, participants completed the demographic/experience questionnaire, followed by the actual experimental tasks that was to comprehend the source-code snippets.

To implement the study, we used SoSci survey[2], in which we integrated REyeker (Mucke et al., 2021). Due to the pandemic, this was the best option within the regulations of our university to conduct the experiment, and in future studies, we intend to replicate this study with an on-site eye tracker to increase the temporal resolution and spatial accuracy of the visual-attention data, and to personally discuss subjective views on recursion and iteration with our participants.

Snippet	Variant	Recursive				Iterative			
		Total Responses	Response Correctness in %	Mean Response Time [s] of corrects	Mean number of fixations per participant	Total Responses	Response Correctness in %	Mean Response Time [s] of corrects	Mean number of fixations per participant
Binary Search	Bottom-up	7	85,71%	441	98	12	16,67%	283	193
	Top-down	13	23,08%	202	49	15	53,33%	322	72
Bubble Sort	Bottom-up	25	52,00%	260	38	23	30,43%	261	42
	Top-down	26	65,38%	27	85	29	51,72%	401	78
Factorial	Bottom-up	25	88,00%	132	27	29	89,66%	139	22
	Top-down	25	96,00%	77	22	24	75,00%	109	24
Fibonacci	Bottom-up	24	50,00%	180	32	24	70,83%	240	53
	Top-down	28	53,57%	200	37	25	88,00%	180	74
Integer to Binary	Bottom-up	13	46,15%	420	38	8	75,00%	399	81
	Top-down	14	78,57%	130	13	13	92,31%	210	15
Multiply Matrix	Bottom-up	11	36,36%	550	48	10	70,00%	464	50
	Top-down	9	44,44%	596	187	12	25,00%	256	136
Prime Factors	Bottom-up	15	20,00%	142	27	14	21,43%	510	57
	Top-down	13	30,77%	315	62	11	45,45%	166	135
Reverse String	Bottom-up	26	53,85%	217	59	26	80,77%	224	53
	Top-down	23	52,17%	250	27	25	88,00%	160	47
	Bottom-up	146	54,79%	241±256	46	146	60,96%	250±241	69
	Top-down	151	59,60%	161±207	60	154	68,18%	212±238	73
	Total	297	57,24%	199±235	53	300	64,67%	230±239	71

Table 3 – All snippets used in the study, their corresponding correctness in % and response time and mean number of fixations per participants for iterative and recursive programs.

3. Results

In this section, we present the results separately for each research question.

3.1. RQ1: Behavioral Data

Preparation and Pre-Processing At the beginning of the pre-processing of the behavioral data, we evaluated the correctness of the responses. We considered responses with only formatting inaccuracies as correct (e.g., in the matrix multiplication algorithm, some participants responded with '6 6 6 \n 12 12 12 \n 18 18 18' and others responded with '{{6 6 6}{12 12 12}{18 18 18}}'). We used regular expressions for this purpose. For example, the regex for the matrix multiplication is '.*6.*6.*6.*12.*12.*12.*18.*18.*18.*'. Based on this procedure, we evaluated 389 out of 597 answers as correct. The next step was outlier removal, in which we removed all wrong responses. The reason is that we intended to filter out participants who quickly went through the experiment to gain extra points for the course. This way, we ensured to only analyze high-quality data. Since we used a within-subject design, the deletion of outliers would lead to fractional data for some conditions. To mitigate this problem, we applied the following procedure, depending on the number of deleted cases per participant:

- One or two outliers per participant: If we only needed to delete one or two data points per participant, we filled in the mean of the according condition(s) for that participant. Say, for Participant X, we have a response time of 29 seconds and a wrong answer for the iterative top-down condition. In this case, we replace the value with the mean (i.e., 241). This happened for one participant.
- Three or more outliers per participant: With values missing for 3 or 4 conditions, we removed the according participant's data from the analysis (7 participants' data).

Descriptive Statistics Table 3 summarizes the behavioral data, separated by implementation style, comprehension strategy, and algorithm. The number of responses differs, as not all participants saw all algorithms (but all conditions).

We discuss RQ1 in two parts: response correctness followed by response time. We visualize the correctness ratios in Figure 4 for all the algorithms and the total of the conditions to provide an overview of the data. The response correctness ratio is the highest in iterative top-down versions. However, all correctness ratios are roughly in the same magnitude. Interestingly, the data show enormous differences in

[2](Leiner, 2014), available at `https://www.soscisurvey.de/`

the response correctness ratios across algorithms. For example, binary search shows a difference in the two styles that is the top-down recursive has a lower correctness (23.08%) than the top-down iterative variant (53.33%). In contrast, matrix multiplication has a higher correctness in the iterative bottom-up implementation (70%) than the recursive counterpart (36.36%). Likewise, for the comprehension strategies, some algorithms have a higher correctness in top-down comprehension, for example, bubble sort, while the algorithm integer to binary has a higher correctness in overall bottom-up comprehension.

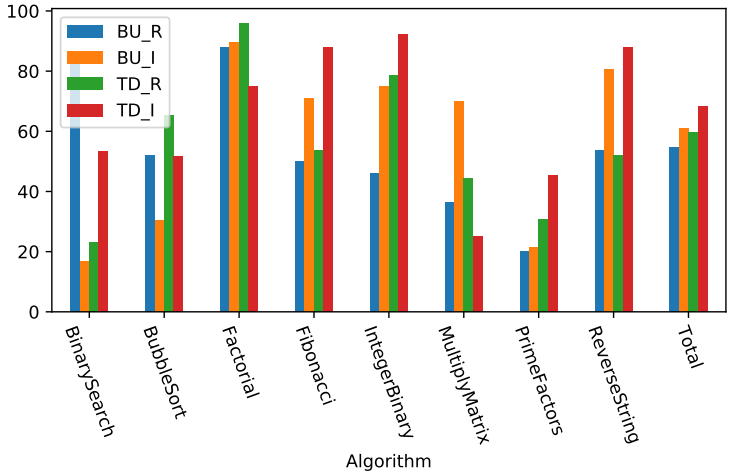


Figure 4 – Participants' correctness ratio in % for iterative and recursive programs, categorized by top-down and bottom-up comprehension.

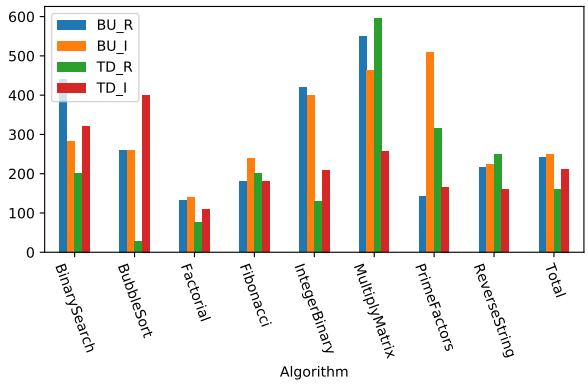


Figure 5 – Participants' response time for iterative and recursive programs, categorized by top-down and bottom-up comprehension.

There are some differences in all four conditions. The one that stands out is the top-down recursive variant, in which the mean response time is lowest, illustrated in Figure 6. Beside having the lowest mean, the top-down recursive variant also has the smallest variance. On the other side of the range is the bottom-up iterative condition, which has the highest mean of all the conditions. This is not consistent across all algorithms. For example, the bottom-up recursive prime factors (mean of 142) implementation was understood faster than the recursive top-down (mean of 315), which is illustrated in Figure 4.

Inferential Statistics To evaluate whether there is a significant difference in the correctness ratio, we performed a chi-squared test. We found no statistically significant difference in the correctness of iterative and recursive styles of programs with different comprehension strategies (χ^2 = 5.3402, df = 3, p-value = 0.1485).

We analyzed the statistical significance of the response-time data with a two-way ANOVA with repeated measures. We present the results in Table 4. We conduct the ANOVA for completeness, but instead of

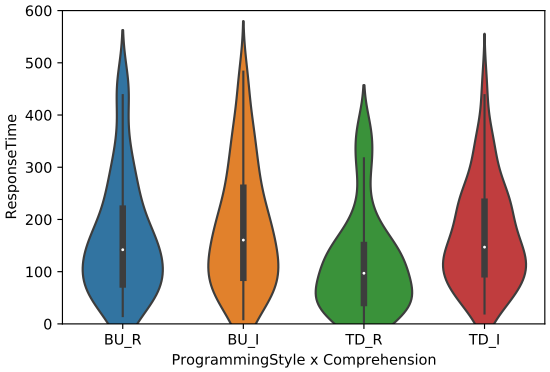


Figure 6 – Participants' response time in seconds for iterative and recursive programs, categorized by top-down and bottom-up comprehension.

Main Effect	F value	Pr(>F)	η^2
ProgrammingStyle	1.5631	0.21197	4.04e-03
ComprehensionStrategy	5.9104	0.01551	0.02
ProgrammingStyle:ComprehensionStrategy	0.7740	0.37953	2.01e-03

Table 4 – Results for two-way ANOVA and η^2 test with repeated measures.

looking at significance, we assess the effect size, that is η^2. Even though we find a significant effect of comprehension strategy, we are reluctant to treat it as a true effect, because the effect size is close to 0.

Answer to RQ1: Overall, we found no indication that programming style or comprehension strategy affect students' performance.

3.2. RQ2: Visual Attention

Preparation and Pre-Processing REYEKER data do not require any further pre-processing. For visualization, we use only data of correct responses.

Descriptive Statistics We analyzed the visual-attention data that REYEKER had generated. REYEKER can produce several visualizations, including heat maps, chronological sequences of click data, and transitions between areas-of-interest (AOIs). For this experiment, we focused on heat maps and transitions between AOIs.

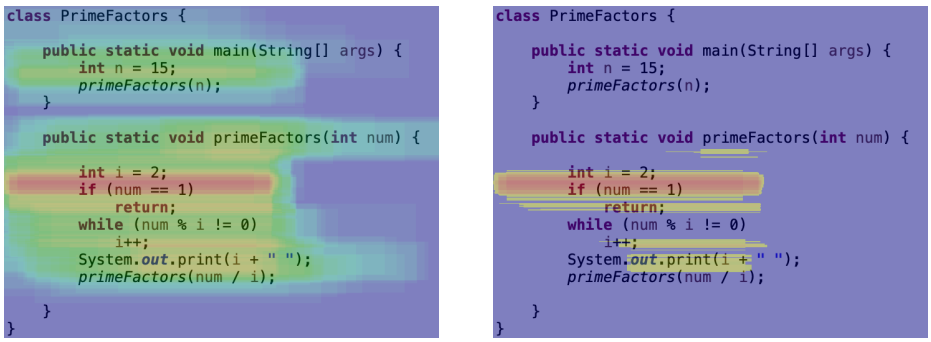
In particular, we have generated average heat maps for all 32 code snippets that were comprehended by participants. In Figure 8, we provide an overview of all heat maps based on programming style and comprehension type.

For data analysis, we defined the following AOIs, which depend on the semantics of the code snippets:

1. 'main' - Code in the main method.
2. 'Declaration' - Declaration of the non-main method in the snippet.
3. 'Initialization' - Initialization done before any recursive or iterative specific pattern.
4. 'Programming Style Condition' - Condition for a loop in case of iteration or the termination condition in case of recursion.
5. 'Programming Style Step' - Calculations in the loop for iterative algorithm or calculations done with the recursive function.
6. 'Return Result' - The source code for returning the result, either as return or side effect via console output.

For an explicit categorization of the heat maps, we first defined what the focus points are with respect

to the measured “heat”. We set a threshold of 70 %, meaning everything below 70 % max heat is not a focus point. For example, Figure 7 displays a comparison of heat maps for the PrimeFactors (R_TD’) snippet generated before and after we set the threshold of 70 %. To agree on a threshold, parts of the author team annotated each heat map to decide what a hot spot is, and the 70 % was the interpersonal consensus. We document all threshold-applied heat maps on the project website.



(a) (b)

Figure 7 – Example of the differences between heat maps generated before 70% threshold applied (a) vs after 70% threshold applied (b)

Based on the predefined AOIs, we visualize how the visual attention of the participants transitioned between AOIs. All measures of visual attention were assigned to an AOI, and two consecutive measures of click data became a transition.

In Figure 9, the width of the sections in a diagram represents how often an area of interest was clicked on, the lines and their width between the areas show which transitions were used how often. In between the comprehension strategies of the algorithms, there is just a small difference regarding the transitions. As shown in the iterative top-down and iterative bottom-up diagram, the circle arcs have approximately the same size. This also holds true for top-down recursive and bottom-up recursive. However, iterative vs. recursive diagrams reflect that the declaration of the method was visited more often in recursive programs than the iterative ones. Similarly, programming style condition has more visits in iterative conditions. For the transitions in the recursive diagrams, a strongly connected line between declaration of the method (area 2) and the recursive step (area 5) is visible. This transition is not pronounced in the iterative variant.

We found no evident differences in the heat maps between the two comprehension strategies.The main cluster of visual attention appears to be the computing method, unless it is complex input data that is being manipulated (e.g., bubble sort with an array of 7 numbers, binary search with 5 numbers, or matrix multiplication). Algorithms with complex input show focus points in the main method, because students may not be able to keep the total input in their working memory. Therefore, students must revisit the input, which leads to the hot spot in the main method.

Based on the transition diagrams, there does not seem to be a difference between top-down and bottom-up comprehension. This could be because our participants were relatively new to programming and have not yet developed the necessary experience to elicit top-down comprehension. Another factor could be, that the students were unfamiliar with the presented algorithms, and therefore top-down versus bottom-up strategy had no influence on their comprehension behavior. But, for the recursive versus iterative conditions some differences are noticeable, such as how often some AOIs were visited and how the students transitioned between them. For example, the declaration was looked at way more often in recursive programs, this could be, because the declaration of the recursive method is important for the function itself, while in iterative programming setting up the loop is more important for the comprehension, one could carve this out from the bigger influence of the "Programming Style Condition". Another prominent part in the transition diagrams were the transition between those AOIs, in the recursive diagrams

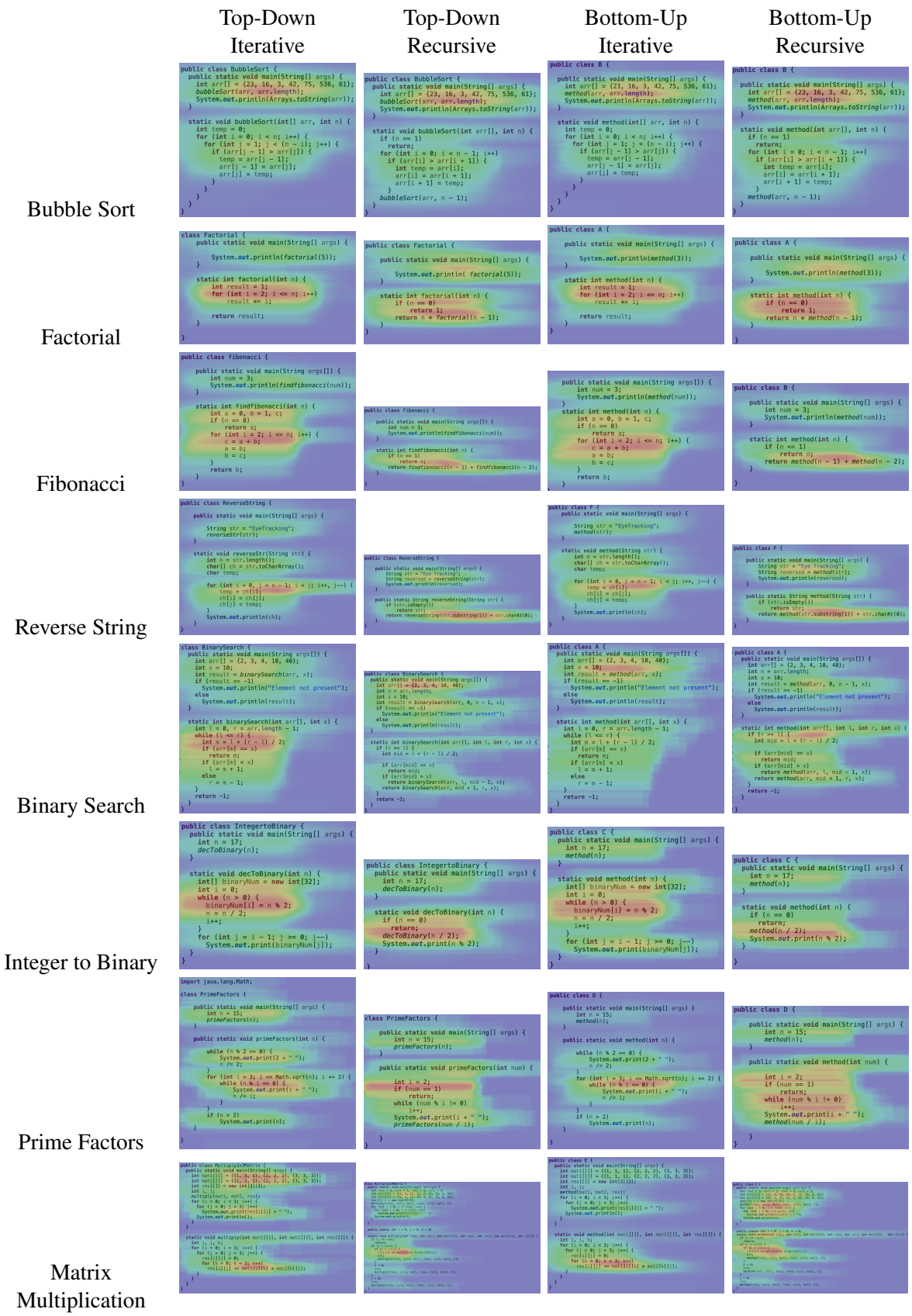
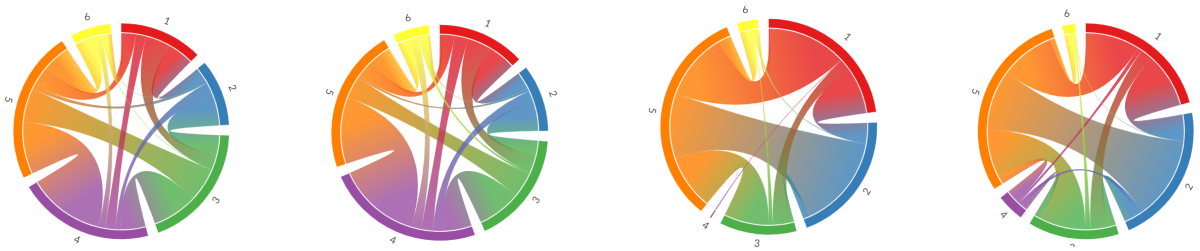


Figure 8 – Heatmap for all programs of the four variations.



(a) Iterative Top-Down *(b) Iterative Bottom-Up* *(c) Recursive Top-Down* *(d) Recursive Bottom-Up*

Figure 9 – Illustration of various areas of interest and the transition between these areas.In above Figures Area 1 is 'main'. Area 2 is 'Declaration'. Area 3 is 'Initialization'. Area 4 is 'programming Style Condition'. Area 5 is 'Programming Style Step'. Area 6 is 'Return Result'

one can see a big connection between the "Declaration" and the "Programming Style Step", which indicates a strong connection between the method declaration to the recursive call of the function, which follows the control flow of the code. This closeness also exists for the iterative programming where the iterative condition (e.g., the loop) and the iterative step (e.g., the code in the loop) have a strong connection in the transition diagrams.

Answer to RQ2: Overall, the visual attention was similar between top-down and bottom-up comprehension. Iterative and recursive programs differ in regard to their key areas of interest. Still iterative and recursive programs seems to be scanned through to some degree equally, if one compares the visual attention to the control flow of the source code.

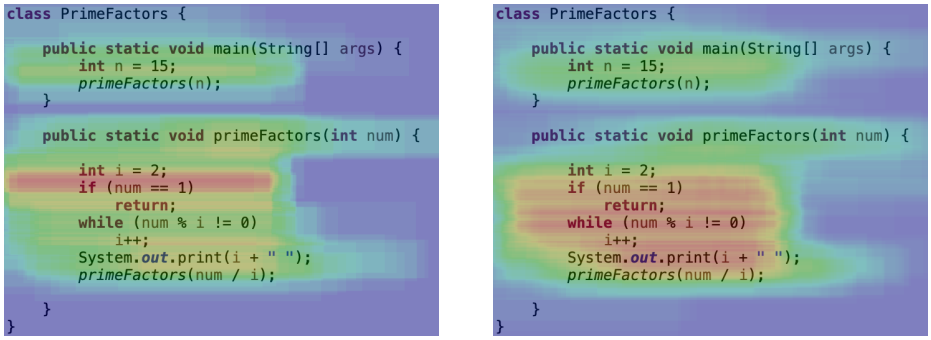
4. Data Exploration

During data analysis, we made several interesting observations that we describe next. Since these are not covered by our research questions, we clearly separate answering the research questions from data exploration.

Based on the heat maps, we found no specific pattern that explains the classification of the several algorithms with regard to patterns defined below. We found that most of the programs have a focus point on main method and calculations, however, we also observed that programs with complex input drives visual attention of the participants.

We extracted the following patterns based on the observed focus points of the heat maps:

- {1} main: Participants focus on the main method more than any other part of the snippet. We observed this in multiple snippets, for example, BubbleSort R_TD, I_BU and R_BU. This pattern was also observed in BinarySearch R_TD and I_BU.
- {1, 4, 5} main + condition + calculation: The main areas of focus for participants were the main method along with the condition statements and calculations. Participants followed this pattern in the matrix multiplication algorithm I_TD, R_TD, and R_BU. We also observe this pattern in Binary Search R_TD.
- {1, 5} main + calculation: Participants' focused on the main method and calculations in the loop for iteration or calculations done with the recursive function. In I_TD BubbleSort, we observe almost the same pattern as in other conditions for BubbleSort algorithm, however, there was an additional focus point on the calculations.
- {3, 4, 5} initialization + condition + calculation: Participants focused on the initialization of the method, the condition and the calculation itself. This is done for the I_TD and I_BU Fibonacci, the I_TD and I_BU Integer to Binary and for the R_TD and R_BU Prime Factors algorithm.
- {4, 5} condition + calculation: Participants focused mostly on the condition of the programming style and the calculation itself. This happened for I_TD, I_BU Factorial, I_TD, I_BU Reverse



(a) (b)

Figure 10 – Example of the differences between heat maps generated from correct (a) vs incorrect (b) responses.

String, R_BU Binary Search, I_TD, I_BU Prime Factors and for the I_BU Matrix Multiplication.

- {5} calculation: Participants' focused mostly on the calculation of the algorithm itself. This was the case for R_TD, R_BU Fibonacci and the R_TD, R_BU Integer to Binary.

Furthermore, we noticed a difference between correct and incorrect responses, which we illustrate exemplary in Figure 10. When comparing heat maps of correct to heat maps of incorrect responses, many of the focus points shifted to the computing method for the incorrect responses. This seems to be independent of programming style and comprehension strategy. This indicates that participants with incorrect answers require more effort in the computing method to comprehend the snippet, or that they used bottom-up comprehension and failed to fully understand a snippet.

These observations can serve as hypotheses that shall be investigated with dedicated experiments. A future study shall aim to identify what helps students to understand unknown programs, and how to improve the correctness of bottom-up comprehension.

5. Interpretation

The collected data suggest that our group of students, who have little experience with programming, understand recursion and iteration equally well. This could be because students were already familiar with both recursion and iteration. Neither recursion, which one might think would be difficult to understand because students have to use backward reasoning to breakdown problems into sub problems until a solution is found, nor iteration, which is perhaps closer to forward reasoning where students start the solution from an initial state leading towards a goal state, appeared to make a difference in how students' understand source code. Instead, participants had equal understanding of both styles (cf. Table 2) and thus have not yet built up a preference through a programmer's everyday life. We observe that iterative programs were no more difficult to comprehend than recursive programs. We also observe that familiarity with the mathematical functions and common algorithms yields higher correctness percentage (e.g., factorial, Fibonacci).

Likewise, it does not seem to make any difference whether top-down or bottom-up comprehension strategy was used, which could again point to a lack of experience. As shown by Soloway and Ehrlich (Soloway & Ehrlich, 1984), even experienced programmers become as slow as beginners when programming conventions are broken. It could be argued that this knowledge of conventions is not yet ingrained in novice programmers, which is why comprehension strategies did not cause a significant difference in the overall comprehension process.

Nevertheless, it is noteworthy that distinct algorithms are comprehended quite differently across the conditions of our study. This may indicate that these algorithms were already known to the beginners in the iterative or recursive variant, or even that these algorithms are generally easier to understand in one of the different programming styles. There are also differences between the comprehension

strategies. Since bottom-up algorithms were generally answered correctly more often, albeit statistically insignificantly, this could indicate that familiar mathematical formulas, such as the Fibonacci numbers, were taken too easy by students, by not considering the starting condition of the respective algorithm. In order to exclude these factors or to look at them more closely, further studies are needed that focus on analyzing these individual factors.

6. Threats to Validity

6.1. Internal Validity

There are several threats to validity arising from our study. We selected programs from previous studies that were of comparable complexity to exclude an influence of the programs themselves. Additionally, we randomized the order of the programs. However, as evident in the presented heat maps, it clearly still has a major influence. This might have overshadowed possible effects of programming style and comprehension strategy.

Our operationalization of visual attention with REYEKER is only an approximation of the natural reading behavior during program comprehension, since the necessary mouse clicks are slower than normal fixations. However, similar to widely accepted think-aloud protocols, we do not expect a substantial bias in favor of only one experiment factor leading to false results, as REYEKER was build on BUBBLEVIEW, which showed a good approximation of visual attention (Kim et al., 2017).

6.2. External Validity

In the presented study, we used a restricted set of iterative and recursive programs that were of fairly basic nature, and we recruited only rather novice programmers. Our results apply only to similar circumstances and cannot easily be generalized to other circumstances, such as complex types of recursion or experienced programmers. Nevertheless, our setting is an important representation, as especially novice programmers start with basic programs.

7. Conclusion

For novice academic programmers, there is no difference in efficiency in understanding fairly low-complexity iterative and recursive algorithms. This includes both the correctness and the response time of the comprehension. Also, there is no significant difference between top-down and bottom-up comprehension with novice programmers.

The visual attention of that same group is equally indistinguishable between top-down and bottom-up implementations. But it seems that the different programming styles have a different approach to understanding the source code, so other beacons in the code become more relevant and these are visited more often in the course of the comprehension process. But to investigate this further, it would be necessary to design the code snippets differently to achieve a better separation between the individual areas-of-interest.

Our results demonstrate the programming style does not have any significant effect on how students understand programs. Furthermore, we identify a need for further studies. Our setting was very limited so that we could not assess cognitive load (e.g., with pupil dilation), follow the students' process when understanding snippets (e.g., by using think-aloud protocol), or use more advanced snippets (e.g., with mutual recursion). We intend to extend our setting once the pandemic situation allows us.

8. References

Anzai, Y., & Uesato, Y. (1982). Learning recursive procedures by middleschool children. In *Proceedings of the fourth annual conference of the cognitive science society* (pp. 100–102).

Barjaktarovic, M. (2012). Teaching mathematics and programming foundations early in curriculum using real-life multicultural examples. In *Proceedings of the international conference on frontiers in education: Computer science and computer engineering (fecs)* (p. 1).

Benander, A. C., Benander, B. A., & Pu, H. (1996). Recursion vs. iteration: An empirical study of comprehension. *Journal of Systems and Software*, *32*(1), 73-82.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, *18*(6), 543-554.

Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J. H., Schulte, C., ... Tamm, S. (2015, May). Eye movements in code reading: Relaxing the linear order. In *Proc. int'l conf. program comprehension (icpc)* (p. 255-265). IEEE.

Dunsmore, A., & Roper, M. (2000). *A Comparative Evaluation of Program Comprehension Measures* (Tech. Rep. No. EFoCS 35-2000). Department of Computer Science, University of Strathclyde.

Felleisen, M., Findler, R. B., Flatt, M., & Krishnamurthi, S. (2018). *How to design programs: An introduction to programming and computing*. The MIT Press.

Gal-Ezer, J., & Harel, D. (1998, 09). What (else) should cs educators know? *Commun. ACM*, *41*, 77-84. doi: 10.1145/285070.285085

Ginat, D. (2005). The suitable way is backwards, but they work forward. *Journal of Computers in Mathematics and Science Teaching*, *24*(1), 73–88.

Kessler, C. M., & Anderson, J. R. (1986, June). Learning flow of control: Recursive and iterative procedures. *Hum.-Comput. Interact.*, *2*(2), 135166.

Kim, N. W., Bylinskii, Z., Borkin, M. A., Gajos, K. Z., Oliva, A., Durand, F., & Pfister, H. (2017). BubbleView: An Interface for Crowdsourcing Image Importance Maps and Tracking Visual Attention. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 36.

Leiner, D. (2014). *Sosci survey (version 3.2.27)[computer software].*

Lye, S., & Koh, J. (2014, 12). Review on teaching and learning of computational thinking through programming: What is next for k-12? *Computers in Human Behavior*, *41*, 5161. doi: 10.1016/j.chb.2014.09.012

Mccauley, R., Grissom, S., Fitzgerald, S., & Murphy, L. (2015, 01). Teaching and learning recursive programming: a review of the research literature. *Computer Science Education*, *25*, 37-66.

Michaelson, G. (2015). Teaching recursion with counting songs. *ACM SIGCHI, June*.

Mucke, J., Schwarzkopf, M., & Siegmund, J. (2021). REyeker: Remote Eye Tracker. In *Proc. Workshop on Eye Movements in Programming (EMIP).* ACM.

Peitek, N., Siegmund, J., & Apel, S. (2020). What drives the reading order of programmers? an eye tracking study. In (p. 342353). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3387904.3389279

Pennington, N. (1987). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, *19*(3), 295–341.

Rinderknecht, C. (2014, 04). A survey on the teaching and learning of recursive programming. *Informatics in Education*, *13*, 87-119.

Roberts, E. S. (Ed.). (1986). *Thinking recursively*. USA: John Wiley & Sons, Inc.

Rubio-Sánchez, M., Urquiza-Fuentes, J., & Pareja-Flores, C. (2008, June). A gentle introduction to mutual recursion. , *40*(3), 235239.

Shriram, K. (2020). *How Not to Teach Recursion.* Retrieved 2021-01-30, from `https://parentheticallyspeaking.org/articles/how-not-to-teach-recursion/`

Siegmund, J., Kastner, C., Apel, S., Parnin, C., Bethmann, A., Leich, T., ... Brechmann, A. (2014). Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th international conference on software engineering* (p. 378389). New York, NY, USA: Association for Computing Machinery.

Siegmund, J., Kästner, C., Liebig, J., Apel, S., & Hanenberg, S. (2014). Measuring and modeling programming experience. *Empirical Softw. Eng.*, *19*(5), 1299–1334.

Siegmund, J., Peitek, N., Parnin, C., Apel, S., Hofmeister, J., Kästner, C., ... Brechmann, A. (2017). Measuring Neural Efficiency of Program Comprehension. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)* (pp. 140–150). ACM.

Soloway, E., & Ehrlich, K. (1984). Empirical Studies of Programming Knowledge. *IEEE Trans. Softw. Eng.*, *10*(5), 595–609.

Sulov, V. (2016, December). Iteration vs recursion in introduction to programming classes. *Cybern. Inf. Technol.*, *16*(4), 6372. Retrieved from https://doi.org/10.1515/cait-2016-0068 doi: 10.1515/cait-2016-0068

Tsaramirsis, G., Al-Jammoor, S., & Buhari, S. (2014, 01). Proposing a new hybrid controlled loop. *International Journal of Software Engineering and its Applications*, *88318*, 203-210. doi: 10 .14257/ijseia.2014.8.3.18

Turbak, F., Royden, C., Stephan, J., & Herbst, J. (2001, 09). Teaching recursion before loops in cs1.

Winslow, L. E. (1996, September). Programming pedagogya psychological overview. *SIGCSE Bull.*, *28*(3), 1722.

Wu, C.-C., Dale, N. B., & Bethel, L. J. (1998, March). Conceptual models and cognitive learning styles in teaching recursion. *SIGCSE Bull.*, *30*(1), 292296.