LogicVis Project Report

CSE 403 Software Engineering
Winter 2019

Team LogV: Candice Miao, Leo Gao, Jed Chen, Glenn Zhang, Andrew Liu

1. Motivations and Objectives

Coding can be difficult to learn. As students, we struggled with many of the fundamental coding concepts. Looking back at these experiences, we wish to build a tool that could make learning how to code much easier as our project. An effective method we found put code into the form of a control flow graphs. Beginner students are used to reading and processing information mostly linearly, and did not have much experience reading code, which contains sequences of steps that jump a lot. This method helps students understand code by connecting to the their prior knowledge of reading flow charts and diagrams[1]. Though graphical tools exist, we noticed a distinct trend amongst all the tools we checked, like JavaVisualizer, and InfoVis Toolkit--they do not handle recursive cases well [9, 10]. We found this to be quite odd because we remembered that recursion was one of the most difficult topics to master, yet it is left out in most tools that are built to help beginners. Through research, we found that the major difficulties students have with recursion are [2, 3]:

Students not utilizing functional abstractions

The purpose of abstractions is so that we can work with something without knowing how it works. For example, there are two forms of knowledge associated with learning how to drive a car: (1) knowing how to operate the car (2) knowing how the car operates. Most drivers do not actually know in detail how cars work, but they can still utilize cars in those ways. In the context of recursion, students have similar troubles: they focus on *how* the recursive step works instead of *what* the recursive step does [2]. If we can help them accept that the recursive step just does what the function itself does, the difficulty of understanding the recursive part of the function diminishes.

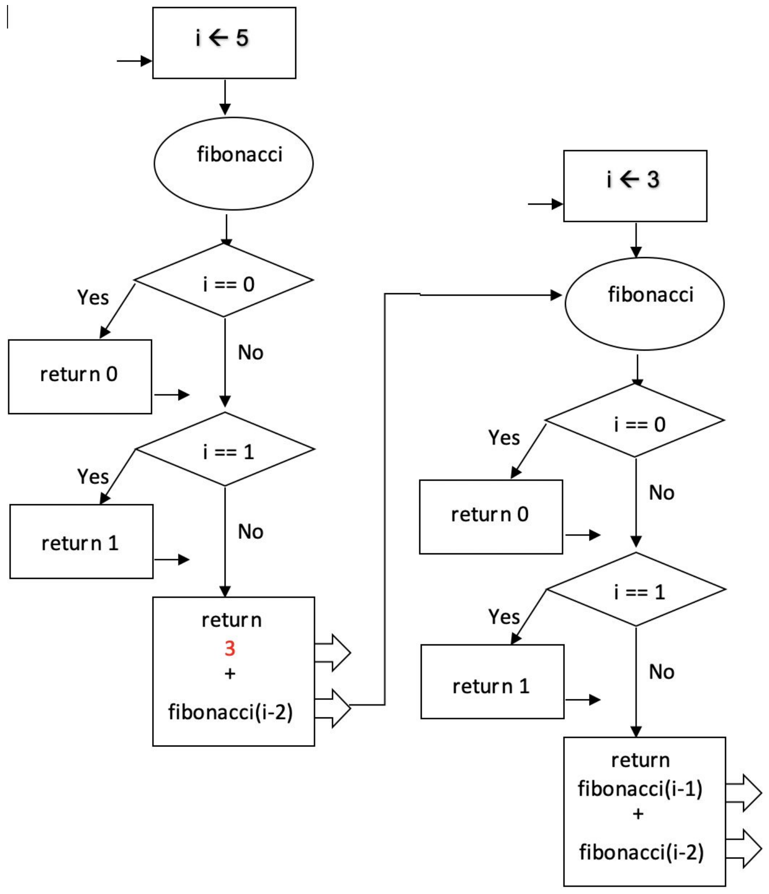
Lack of a proper methodology to represent a recursive solution

Many students that are new to recursion do not understand how to formulate a solution that uses recursion. They begin to write recursive solutions when prompted but often cannot imagine the solution they drafted in a concrete way. For example, they will think of including a base case and a recursive call, but cannot connect these components together to form a coherent solution.

The goal of our project is to build a tool for Java that can help new programmers learn recursion by visualizing the flow behind programs using arrows to connect subsections of the code. This tool can translate a chunk of recursive code to a flowchart so that it is easier for people to grasp the recursive algorithm and build their own recursive programs. In addition to using this tool on existing code, students can also draft their own solutions and use this tool to check whether their solutions behave as intended. Because recursion is a difficult subject, we also design this tool to be used as a supplement in addition to instructions by the instructors teaching.

2. Our Approach

LogicVis is a program that takes an input of a Java recursive function code and its arguments and outputs a logic graph of control flow. It reads in the Java code line by line and turns each line into a node representation with the shape indicating the type of command. LogicVis graphs contain different representations for basic Computer Science concepts, including if/else, loops, recursion, etc. This tool will output the logic graph with input and output parameters tracking for each iteration. As previously mentioned, students generally have two problems with recursion, and we seek to help students by solving these problems. Below is the our graph representation of a simple recursive program.



The following sections will discuss how this approach is expected to combat typical problems students have with recursion.

Not utilizing functional abstractions

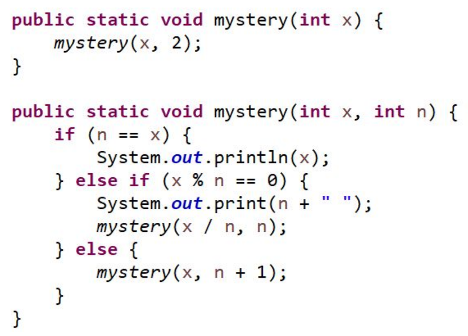
Our approach to this problem is to attempt to unpack the abstractions that is packed in the recursive function. Here we are building our graph to represent those recursive steps concretely. Our take on the visualization of a recursive function exhaustively lists iterations of recursive calls. If the students can see the results of the code by expanding the graphs, they have the potential to acknowledge and trust that the recursive call will perform its intended function. If students are able to accept this idea, they can recognize *what* the calls do without focusing on *how* it is achieved. This does not directly teach students the concept functional abstractions, but it makes recursion more easily recognizable, and by that familiarize student with abstractions. The extra information on the return values and parameters of each call is also here to help students confirm their understanding of the program.

Lack of a proper methodology to represent a recursive solution

This graph also aims to be the alternate methodology for students to represent the codes that they write. Tracing the steps in a recursive program provides students with a mechanical means to follow the recursive algorithms. In our implementation, we will have made it clear that another function frame has been made every time we descend down the stack. Another utility this graph seeks to focus is on helping to debug their own code. Since students are unfamiliar with drafting solutions to a program, intended use of this program is to represent what the students write themselves into concrete graphs. Though this tool does not focus on individual lines, this tool does point out clearly where each line will take place with the nodes of the graph. This way, students will understand the placements of every line they write, as opposed to putting down lines randomly because they imagine they are supposed to be somewhere due to their inability to comprehend recursion.

The recursion will be handled by adding options to expand recursive calls when they appear in the original function graph. They will be generated on demand and allowed a complete expansion in concrete cases. Our minimum viable product will be designed such that a strict number of recursion iterations will be enforced with a working GUI. For the final product, we are aiming for removing the strict limit on the number of recursion iterations but rather utilize the lazy evaluation approach based on scrolling in the UI.

3. Paper Prototype User Study



```
public static void mystery(int x) {
    mystery(x, 2);
}

public static void mystery(int x, int n) {
    if (n == x) {
        System.out.println(x);
    } else if (x % n == 0) {
        System.out.print(n + " ");
        mystery(x / n, n);
    } else {
        mystery(x, n + 1);
    }
}
```

We conducted a user study to confirm that our product will assist with learning recursion. The code for user study is to the left (prints the prime factorization of x). The code is recursive and somewhat complicated. We have developed a paper-prototype for the code here and used it to test the effectiveness of our ideas (See User Manual for UI reference [11]).

3.1 Choice of Method

The tasks given for the users are to compute the results of functions. We split the study into two similar parts: one part without our tool and one part with it. In each part, we will ask similar questions about the program's output and gauge the participants' understanding of the given piece of code. This way, we hope to find whether our tool helps students follow recursion better. We pick this method to mimic methods often used in beginner programming classes while being practical while using a paper prototype.

3.2 Set-up
We performed our study on a college student with a background of AP Computer Science, which means they have learned recursion, but not extensively and not recently. We let the subject look at two methods, hiding both their names (i.e. we labeled them as mystery). The reason we hid the names was to reduce the chances that the participant used their math knowledge to understand the code rather than their knowledge about recursion. We first used a method we showed returned the nth number of the Fibonacci sequence and did provide our paper prototype on the method. We then used a method that printed out a number's prime factorization and included a corresponding paper prototype on the expected product. For each method, we asked the following questions:

- What does factors(20)/fibonacci(5) output? (mystery(20)/mystery(5), to the participant)
- What does this method do?
- How does this method work?

3.3 Performance
The subject was able to identify the output and functionality for both programs without outside assistance. However, the subject spent a few minutes longer on the second prompt, completely missing the base case (went to fibonacci(-1)), and drew a representation of the method calls while computing it. For factors(20), the subject was able to quickly find the output but had trouble explaining why n was incremented by 1 in the second recursive case.

3.4 Feedback
The user responded that, in comparison, our charts did not help understand a specific line of code, but the flowcharts make the execution flow much easier to follow: the flow chart indicates how each recursive step progresses to the next one and helps the user see where each step executes. Furthermore, the user found that there was too much displayed at once; having a reduced flow chart based on which statements execute may make our chart easier to follow.

3.5 Analysis and Future Testing
From the fact that the subject made a separate representation for the first method, we found that the prototype we created successfully served as a separate way of understanding in itself. The representation the subject listed the parameters and return values of each call, which overlaps with and is included by our product's functionality. In addition to being able to characterize the program faster, the subject also had no trouble following the execution of code, likely because all the steps are expanded out for the subject. On the other hand, our prototype failed to give meaning to why each and every line of the code existed.

Further testing about user debugging and developing will be necessary to examine other aspects of this project. These will be performed when more features are available. Follow-ups on this are in the later user studies section.

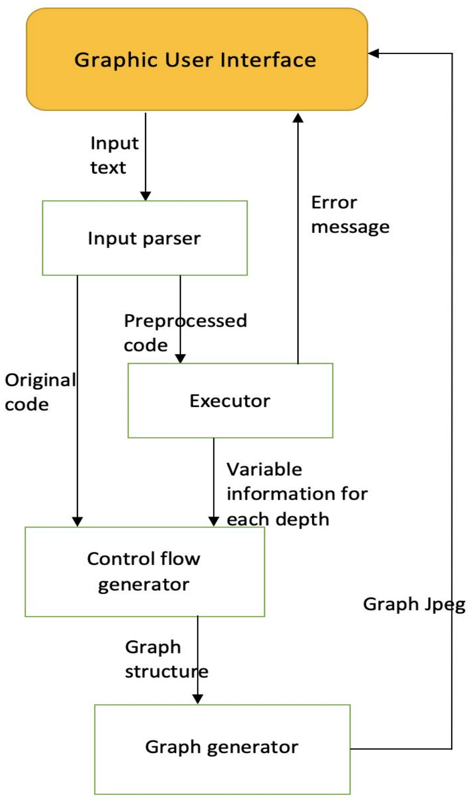
4. Implementation

4.1 Interfaces

(Refer to the user manual [11])

4.2 Architecture

The architecture is represented by the diagram to the left. The boxes represent components of the architecture and the arrows represent the flow of the data from one component to another.



4.2.1 Graphic User Interface

This layer is the front end part of our project. It will let user type input code and get the parameter value and display the graph.

4.2.2 Input parser

This layer will read the input from the front end software, get the input value, preprocess it and send the code to the executor. By processing we mean modifying lines of code in the input text in order to execute it and obtain the desired output in the executor.

4.2.3 Executor

This layer will execute the input code. If the execution failed, send an error message to front end and display error message. If succeeded, the executor will send the code along with the information such as recursion depth and parameter value of each depth to the control flow generator.

4.2.4 Control flow generator

This layer will generate the basic control flow code of each depth of recursion call with corresponding input value and send the control flow graph information to the graph generator. This layer does not generate any graphs, it only generates the data how the graph is supposed to be constructed.

4.2.5 Graph generator

This layer will visualize the graph and implement the clickable features to the graph, then send it back to GUI and display the graph.

4.3 Technologies and Details

We have chosen to develop this software in Java since we are most familiar with the Java language and its libraries.

4.3.1 Graphic User Interface

We use the JavaFX library, which provides a clean graphical UI that works as a standalone. The older libraries such as Swing and AWT do not provide as much functionality whereas others like Pivot are used as RIA. AnchorPane is the layout we decided to use because it gives us more control on putting each element precisely where we want to put it. For specifically the pop-up window, we also used the ControlsFX library. We use a dialog from this library with GridPane to implement our dialog because we need to put a unknown number of elements in the layout and GridPane guarantees the tidiness of the display.

4.3.2 Parser

We utilized the JavaParser API which generates an Abstract Syntax Tree. However, we decided to implement our own tree data structure since the AST has a lot of extra information that we do not need. Also, to increase cohesion, we wanted the parser part to handle tracing to get rid of the extra information rather than having the front end to go through it and draw out the important parts and ignoring other parts. In addition, implementing an easy tree structure doesn't take much time and it will allow us to have more control on what information being carried in the tree structure, we decided to use our own tree structure rather than the API produced AST.

4.3.3 Executor

We build this from scratch. Although we may reference existing flowchart generators, we would have to rewrite most of it if we wanted to use one. The main goal of executor is to take a String of Java code, tell GUI what inputs do we expect from user and based on the inputs user typed in, generate a dynamic graph(not visualizable) which can be updated from a "next" function call. The executor contains three sub parts -- input preprocessor, executor and action generator. Input preprocessor will take a String of Java code as input, analyzes it, then it will inject some lines at specific place in the code in order to get the runtime information after execution. For example, it will add a new parameter int depth that indicates the depth of each recursive call. It also extracts the information such as the parameter types and names and send it to GUI, so that GUI is able to ask user for the input value. Second sub part is executor, the executor used an open source library called beanshell which can execute code on the fly. It will execute the modified code using bean shell and generate a traversable data structure that stores the depth, input value and return value of each recursive call. Last part is an action generator, this part will traverse the data structure from executor and generate a big view of the graph in current graph, the action generator will show the number of control flow graph should be generated, the input values in a specific control flow graph and how each graph is connected to each other. Finally action generator will integrate with Parser and Graph generator and produce a graph that can be updated every time user click "Next".

4.3.4 Graph Generator

There are few java apis that generates flow chart as what we wanted. Therefore, the Graph Generator is implemented from scratch with canvas. To make it easier for generating complicated graph and code reusability, a UIUtil class was implemented that has methods of drawing shapes for all situations.

A graph is generated from the data structure returned by the Parser and Executor. Since such a flow chart is fundamentally a graph data structure, we used a graph data structure to represent the flow chart with their locations calculated and properties specified. After that, the graph is drawn on canvas based on such a data structure. Then replicate the canvas add and modify some inner block information based on the dynamic information given by Executor. The last step is to render the whole canvas to an image so that it can be displayed in a ImageView on the UI.

5. Testing and Results

5.1 Quick lookup of Progress

We are able to produce the frameworks for the UI processes recursive functions. To see these results, go to the root directory of LogicVis' README.md and following the directions about initial results. Our program works under many scenarios, but have a few bugs in them.

Bugs in the code:

- If we have multiple nodes that are on different levels pointing to the same nodes as a child, the arrows will overlap with one another. We have yet to figure out a way to solve it. This may require going through the entire tree once before starting to draw out individual nodes, so they can be located properly.
- We also have a problem dealing the the case when there is statement right after if statement "if (a) return b;" which is a valid java code. We can solve this by adding { } around "return b". So "if (a) {return b;}"

We haven't handle the invalid input yet. If user gives an input code that has run time error or not valid Java code, we should give user instruction on what is problem of current input and we have not finished this part yet.

5.2 Evaluation Methodology

Because this product is focused on assisting students with learning recursion, we determined to use user studies to evaluate our product and determine the way forward. The following sections describe how the studies are performed.

5.2.1 Setup

We find computer science students of varying experiences with coding and ask them to perform programming tasks we will evaluate them on. As we have the complete product, we no longer have the restrictions we do previously from the paper prototype. The problems we pick are of the two types: Fill in the blank and Debugging (The actual code and specification are in Appendix C), which are more difficult and comprehensive than mystery output type questions in the previous experiment. In these experiments, we first introduce the problem given. Our product is given to use in some experiments, but for some trials, we evaluated the the test subjects without our tool as the control group. This is to simulate a situation where an instructor tries to teach a student about recursion using this tool.

5.2.2 Metrics

We measure the performance of the test subjects by timing their time to reach the solution and analyzing the processes made by each subject during that time. Two testers will be present during this process: one provides assistance such as explanations and hints on the questions and one takes notes on the interactions and occasionally provide explanation if the other tester missed some point. This process will conclude once the test subject solves the question.

5.3 Preliminary Results

5.3.1 Restrictions

Due to difficulties in the technical progress at this time, the user studies cannot be conducted exactly as described above. The tool we were provided with only generates the graph for a single iteration of the recursion, rather than being able to expand as we have expected. So, instead of conducting the experiment above, we ran the experiment using the existing tool and let one of the testers generate the arguments and return values of each iteration call by hand. This attempts to simulate the expected product, but is lacking in aspects since the complete graph is not generated. Because the complete effects of changes are not shown, we chose to forgo the fill in the blank question as it does not help students formulate solutions well. We then performed the experiment with the altered setup using the programs odds() and printTwos() (details in Appendix C). We picked these programs because these programs require a good understanding of is going on in the program in order to debug them. For odds(), the user need to distinguish the how the 2 cases of even and odd are handled differently. For printTwos(), the user needs to have a good understanding of how the recursion call is processed and the order at which each print statements are called. Both of these can possibly be made easier with our program, which attempts to show the order of execution.

5.3.2 Results

While the subjects were using the tool, a common complaint is that the elements in the graphs are too difficult to see. Because the control flow of the code given are simple, the low graphics quality graph provided very little assistance. The following are observation of each individual test subject.

Subject Experience	Time Spent	Observations
Almost no Java experience, had Matlab experience (No recursion knowledge)	- 57 minutes to solve odds() without the tool - 32 minutes to solve printTwos() with the tool.	The subject showed a lot of confusion regarding the Java syntax and the concept of recursion in both the control and the tool version. The tool was not able to help much in this case: the program uses print lines for output as opposed to return values. The subject used tool to identify parameters over the iterations, but was not able to utilize the tool beyond that. The reduction

		in the time can be attributed to both the familiarity and use of the parameter calls tool in the second question.
High School Computer Science (Has learned about recursion)	- 24 minutes to solve printTwos() without the tool - 6 minutes to solve odds() with the tool.	The subject expressed inability of understanding the Mathematics behind the programming statements of printTwos(). For odds(), the tool proved helpful by providing the participant with the parameters and return values of each iteration. The large reduction in time is likely due to noticing that handling odds and evens separately solves the problem upon looking at the arguments of each recursive call, which is given by the tool.
Basic Collegiate Programming and Data Structures (Has learned about recursion)	- 27 minutes to solve odds() with the tool	The subject once again had difficulties over the Mathematics properties of the problem. The link between editing the integer string and performing Mathematics operations were very difficult to grasp. The subject also started with misunderstandings of the problem and showed problems with debugging rather than with recursion in general, which extended the time by quite a bit. The subject used the tool but was not able to overcome the misunderstanding until the tester clarified it.

5.3.3 Analysis

The poor UI and limited information in the graph generated decreased the effectiveness of the tool by a lot. However, our tool showed promise in its ability to display return values and argument values in each recursive call. This demonstrates that the tool does serve as a solution to students often not being able to recognize recursion as an abstraction--seeing the results of each call convinces students to trust the recursive calls to perform its function when writing their own code.

From this, we found that our current priority should be to make the UI look better and be able to expand the graph into following call iterations. Further user testing must be conducted once more features of the product are implemented.

5.4 Formal Results

5.4.1 Restrictions

Due to the same technical difficulties at this time, we simplified the experiment again, but this time we have better graphics than last time. Since the tool had a processing issue, the tester had to narrate over what the product should be showing at each step. However, with a more readable UI, the basic graph generation was usable and given to the participant to see, making this process much easier. We performed the experiment with this altered setup using the programs odds() and removeOnes() (details in Appendix C). We picked odds() for the same reason, and we replaced the debugging question printTwos() with the fill in the blank question removeOnes() since it is a more intuitive question with a more interactive approach--The subject needs to compose code in order to solve the problem.

5.4.2 Results

We tested on a single user this time with one control and one tool-assisted problem.

Subject Experience	Time Spent	Observations
Basic Collegiate Programming (Has learned about recursion)	- 20 minutes to solve odds() without the tool - 31 minutes to solve removeOnes() with the tool	The participant had no trouble identifying the bug in odds(). However, he encountered issues when trying to follow what the current method was doing. So, he opted to recreate the method himself. When doing so, the participant struggled to write the correct recursive call, as he was unable to identify where his code was not working. For removeOnes(), it only took a few minutes for the participant to fill in the base case and case when the current digit is one, but he again struggled to identify the appropriate recursion call. With the use of our tool, he was able to identify the bugs that he wrote, and it ultimately led to the solution (after seeing the expected and actual return value in one of the calls).

5.4.3 Analysis

Since our initial tests simulated what we desired our program to do, we did not find much of a difference in our latest test. The most notable difference is that the user got to interact with the graph, which let the user follow the code in a simpler format and make significant improvements in terms of debugging. The step function helped to identify why current iterations of the code had poor output or infinite loops, which ended up leading the user to the proper solution. At the same time, our tool did not help the user abstract the concept of recursion as a program that calls itself to do most of the work. We are alright with that, since we believe this tool should not be used to learn the abstraction but as a way to concretely show what recursion is doing at each step in order to reinforce that the abstraction works. Knowing the concepts of recursion is vital to being able to effectively learn from this tool.

Overall, we consider this evaluation to be somewhat successful, since it did help the user understand their own recursion code. However, since we did not see any improvements in terms of speed as we hoped, we understand that there are shortcomings in how effectively our tool helps write code from scratch.

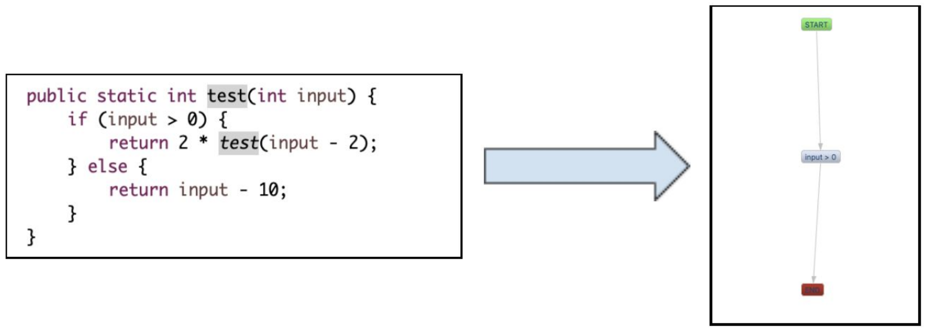
6. Limitations

We run risks that our approach is not received well by the intended audience. The reasons for this may be having unintuitive navigation, unclear information, or distracting factors in the final product. To mitigate the harmful impact of such risks, we have been conducting user research and found ways to avoid some issues according to the feedbacks given. More changes will be necessary.

We also have taken limitations in deciding the scope of our project. We have chosen to make this software specifically for Java and best used as a supplement in addition to traditional ways such as being taught in class. We based this on the idea that most students that learn computer science start with Java and in the setting of a typical classroom. Another risk that we will be taking is our decision to not handle Java built-in library functions. Displaying these would introduce a whole new set of problems, such as additional clutter and having to implement it in the first place. Overall, we hope to build our software for a large enough scope of statements to help the user while not also overwhelming them. If we have successfully chosen the correct scope for our product, we will have made the version of our software that will help most people.

7. Related Work

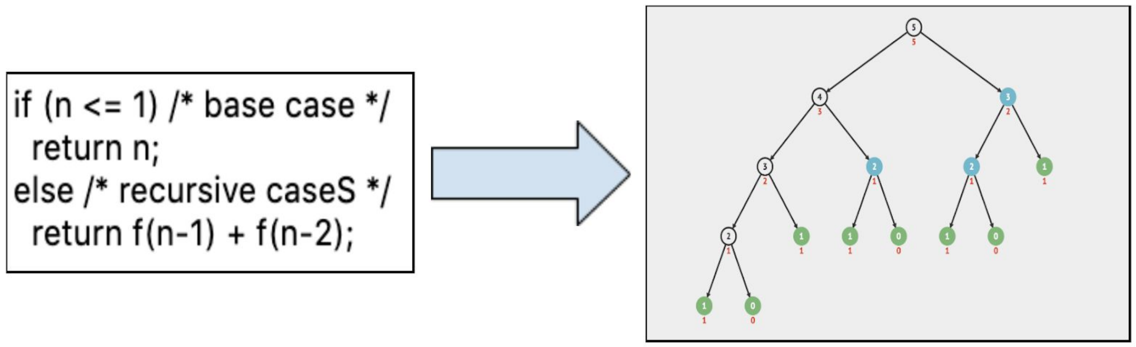
As we mentioned before, there are many visualization tools online, but none of them handle recursion to the amount of detail we hoped for. For example, the Eclipse CFG Generator[4] was designed as an Eclipse plugin to translate a piece of code into a flowchart (See Figure 1). The following graph is an example of how it works. However, the visualization it produces neither shows what each line of code is doing (each line of code are simplified to the category of the code or the name of the variable, like “expression” and “plaintext”) nor it shows a clear logic flow within the code (it only visualizes the if/else condition and ignores all other method calls) which makes it impossible to visualize recursion. Also, the CFG Generator and many other tools[5][6] fall short in that they do not handle recursion any more than simply noting that the method was called. For example, Eclipse CFG Generator simplifies the function call to “expression” which tells no information about this function call. In other words, though it helps users understand a program’s flow in a low level, users will still be unable to understand a recursive piece of code, as it provides no extra help on the abstraction of recursion under the call. In comparison, our approach gives visual representations of every single line of code within the recursive function and we track the parameters and return values of the recursive calls, giving the users of our program, especially students, a better sense of the current iteration of the recursion.



```
public static int test(int input) {
    if (input > 0) {
        return 2 * test(input - 2);
    } else {
        return input - 10;
    }
}
```

(Figure 1: Eclipse CFG Generator)

The best recursion trackers we found online were those like VisuAlgo[7]. VisuAlgo runs a program, such as GCD, and demonstrates a live example of the method call flow as a recursive tree. It lets you step one recursive call at a time and shows the parameter values and return values of each call. See figure 2 for the VisuAlgo example.

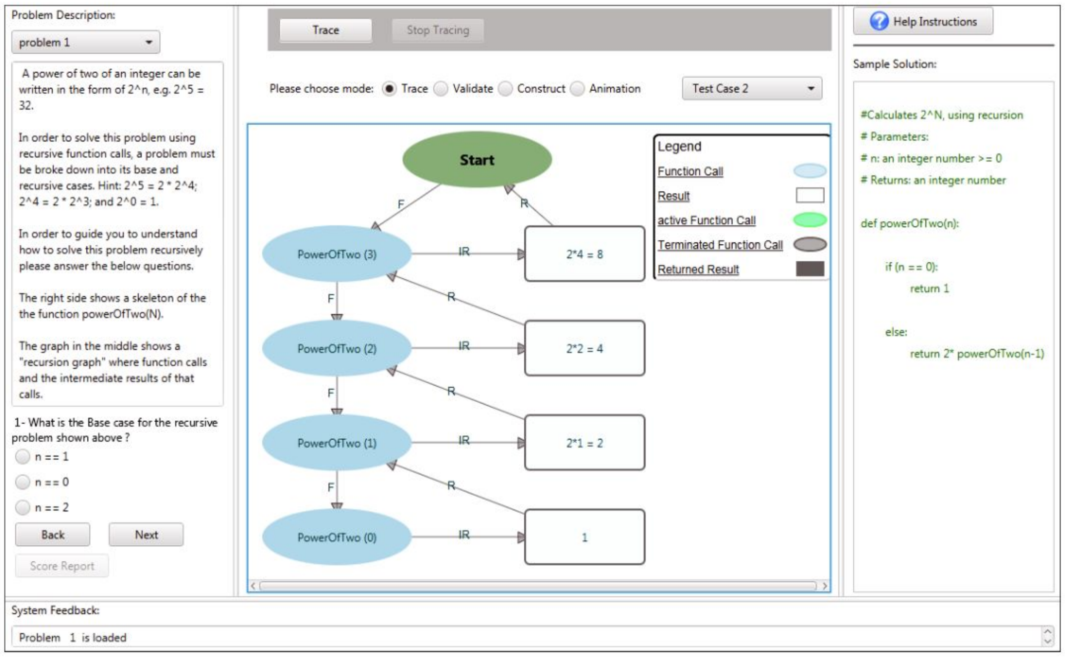


```
if (n <= 1) /* base case */
  return n;
else /* recursive caseS */
  return f(n-1) + f(n-2);
```

(Figure 2: VisuAlgo)

We are using a similar approach, stepping one recursive call at a time with the parameter and return value visualized. The difference is that VisuAlgo represents each recursive calls as a single node with no information on what happens inside the function. Students struggle to understand the recursive program because they have little to no idea about how the result was obtained. We have looked for many tools that track complicated recursive calls, but no such graphing tools exist. Our approach takes into account every line of code, making the user completely aware of the program execution without high level uses of abstractions. In addition, when students use LogicVis to examine their own code, they can contrast our outputs to grasp the impact of each specific line, which VisuAlgo cannot do.

Another approach on helping student learning recursion is ChiQat-Tutor System, a system that helps visualize certain predefined recursive functions[8] (See Figure 3). It is helpful to get students to understand the given recursive function cases by giving students questions and tasks to do about given recursive functions, but it is similar to the previous examples in that they do not generate the details about the recursive call. Since this tool is not able to process non-preset programs, students may also have difficulties with reasoning about their own recursive code during the debugging process. Our approach plans to be able to generalize on all the recursive functions to provide specific graphs on recursive programs for students to compare.


Problem Description:

problem 1

A power of two of an integer can be written in the form of 2^n, e.g. 2^5 = 32.

In order to solve this problem using recursive function calls, a problem must be broke down into its base and recursive cases. Hint: 2^5 = 2 * 2^4; 2^4 = 2 * 2^3; and 2^0 = 1.

In order to guide you to understand how to solve this problem recursively please answer the below questions.

The right side shows a skeleton of the the function powerOfTwo(N).

The graph in the middle shows a "recursion graph" where function calls and the intermediate results of that calls.

1- What is the Base case for the recursive problem shown above ?

- n == 1
- n == 0
- n == 2

Please choose mode: [✓ Trace] Validate Construct Animation — Test Case 2

Legend
- Function Call
- Result
- active Function Call
- Terminated Function Call
- Returned Result

Sample Solution:

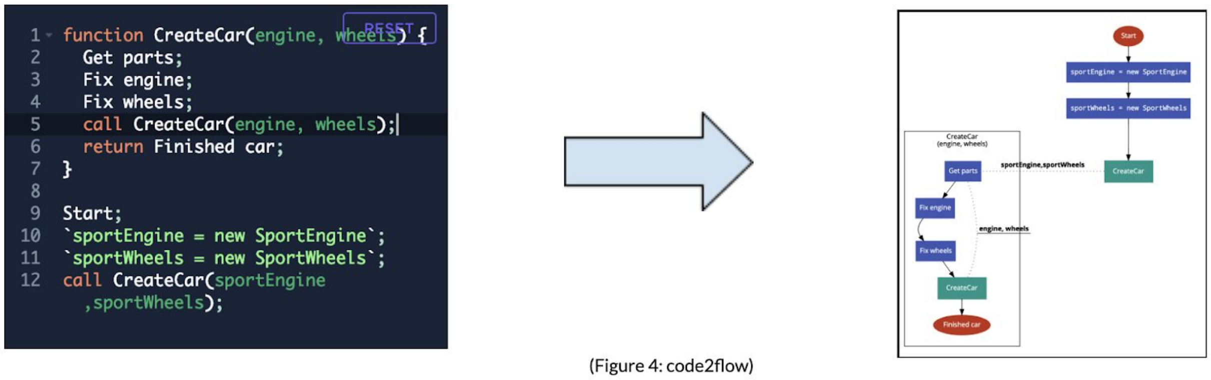
```
#Calculates 2^N, using recursion
# Parameters:
# n: an integer number >= 0
# Returns: an integer number

def powerOfTwo(n):

    if (n == 0):
        return 1

    else:
        return 2* powerOfTwo(n-1)
```

System Feedback:

Problem 1 is loaded

(Figure 3: ChiQat Tutor)

There is one tool that helps code visualization extremely well, which is code2flow[6] (See Figure 4). It takes text input and generates control flow based on it. It also visualizes function calls with details by having a box containing details of the function outside the main graph. The following graph is an example.



```
function CreateCar(engine, wheels) {
  Get parts;
  Fix engine;
  Fix wheels;
  call CreateCar(engine, wheels);
  return Finished car;
}

Start;
`sportEngine = new SportEngine`;
`sportWheels = new SportWheels`;
call CreateCar(sportEngine
  ,sportWheels);
```

(Figure 4: code2flow)

Our approach plans to generate a control flow graph similar to this because it makes each step of the execution clear and easier to understand. However, this tool only gives a static graph with no parameters and return values indicated in the output. In the case of understanding recursion, parameters and return values helps students a lot on understanding the current fields of the execution and the result of each iteration. Our approach enables users to formulate their own code by demonstrating the details of their code based on customized inputs and stepping each iterations of the recursive call to have a better awareness of the overall iterations of the program execution.

8. Conclusion

Throughout our research, our initial motivation has been reinforced: recursion is indeed difficult to understand. The visualization of recursion has helped our research participants solve problems, but it still remains difficult to measure an increase in understanding after using our tool, given our current technical limitations.

Our tool shows promise in the implemented parts just as what we expected it to do. However, some planned components of the product remains buggy. This may hinder some more complicated programs from working as expected in our program.

Hours spent: 50 hours

Appendix A: Schedule

We plan to split major sections of the projects into work done for each week. For each week, we will begin planning the specifics of what each goal entails. After reaching the goal of implementation for every week, we start testing the programs to ensure its correctness. This means that testing will be involved in every week, so we chose to omit it in the schedule below.

Time	Goals
Week 4	- Project proposal and Planning - Write the initial specification - Design visual components of graphs
Week 5	- Finalize the implementation directions and formal proposal document - Assign roles and methods of communication
Week 6	- Design the GUI - Write the user manual - Implement function parsing: Identify the different type of lines in Java - Implement recursion parsing: recognize keywords and represent the flow
Week 7	- Build a data structure to represent the code - Visualize the data structure into a readable graph - Evaluate the performance of the graph using testers
Week 8	- Complete the frame of the UI - Complete processing basic codes excluding recursion and showing initial results
Week 9	- Finish the UI - Finish processing recursion - Research on effectiveness of the tool
Week 10	- Finish the GUI - Link relations between existing functions - Research on effectiveness of the GUI
Week 11	- Final testing - Prepare for presentation

If we have time left over for extra development, we can consider implementing:

- Parsing class structures
- Handling a class with recursive methods

- Making it a plugin for Eclipse
- Supporting modification on generated graphs being reflected in the code

A good “midterm exam” for this project would be checking to ensure that the base visualization of the graph we are building matches our expectations at week 7. This milestone was reached at week 8, and the resulting graphs of the inputs were up to standards. A good “final exam” for this project would be looking at the final product to see whether we are handling multiple functions correctly at the end of week 11. At that point, we would also test everything else to see that they are robust and optimal rather than just its correctness according to the specification.

Appendix B: Roles

Implementation Team: in charge of programming the software so that it runs.

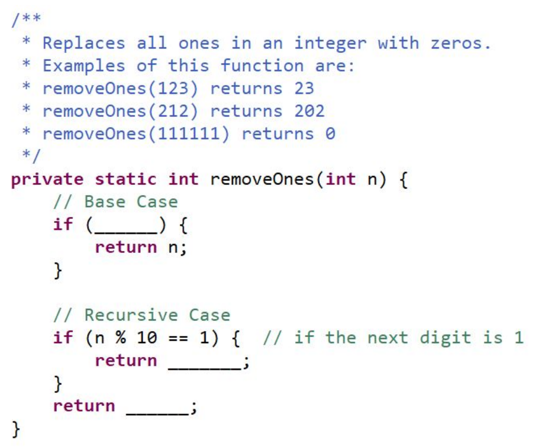
- Andrew Liu
- Candice Miao
- Leo Gao

Evaluation Team: in charge of designing the product and testing its effectiveness.

- Glenn Zhang
- Jed Chen

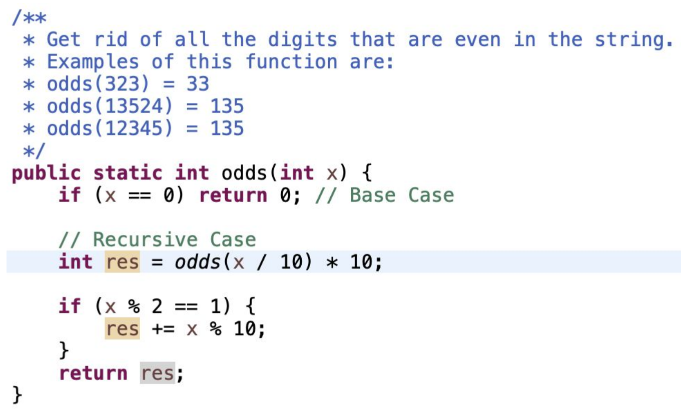
Appendix C: Supplementary Information of User Studies

removeOnes Fill in the Blank question:



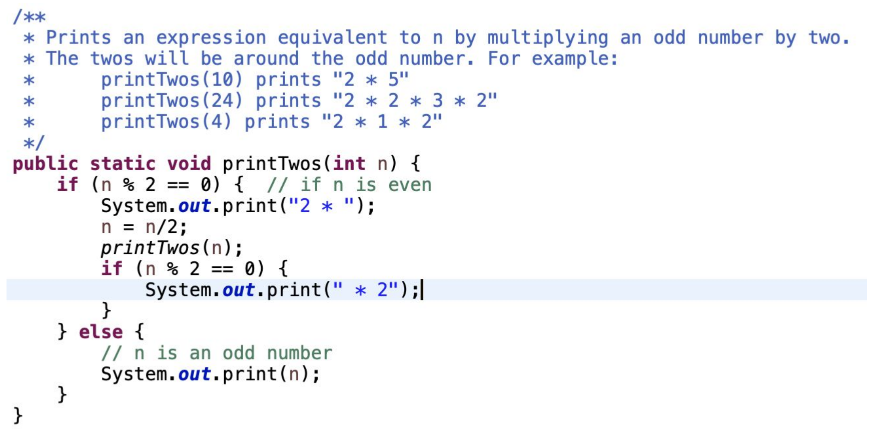
```
/**
 * Replaces all ones in an integer with zeros.
 * Examples of this function are:
 * removeOnes(123) returns 23
 * removeOnes(212) returns 202
 * removeOnes(111111) returns 0
 */
private static int removeOnes(int n) {
    // Base Case
    if (______) {
        return n;
    }

    // Recursive Case
    if (n % 10 == 1) {  // if the next digit is 1
        return _______;
    }
    return ______;
}
```

odds debugging question:



```
/**
 * Get rid of all the digits that are even in the string.
 * Examples of this function are:
 * odds(323) = 33
 * odds(13524) = 135
 * odds(12345) = 135
 */
public static int odds(int x) {
    if (x == 0) return 0; // Base Case

    // Recursive Case
    int res = odds(x / 10) * 10;

    if (x % 2 == 1) {
        res += x % 10;
    }
    return res;
}
```

printTwos debugging question:



```
/**
 * Prints an expression equivalent to n by multiplying an odd number by two.
 * The twos will be around the odd number. For example:
 *      printTwos(10) prints "2 * 5"
 *      printTwos(24) prints "2 * 2 * 3 * 2"
 *      printTwos(4) prints "2 * 1 * 2"
 */
public static void printTwos(int n) {
    if (n % 2 == 0) {  // if n is even
        System.out.print("2 * ");
        n = n/2;
        printTwos(n);
        if (n % 2 == 0) {
            System.out.print(" * 2");
        }
    } else {
        // n is an odd number
        System.out.print(n);
    }
}
```

Citations

[1] Cross II, J.H. & Sheppard, S.V.. (1988). The control structure diagram. 274 - 278. Computers and Communications, 1988. Conference Proceedings., Seventh Annual International Phoenix Conference on. 10.1109/PCCC.1988.10084.

[2] Raja Sooriamurthi. Problems in Comprehending Recursion and Suggested Solutions. University of West Florida, 2001. Proceedings of the 6th annual conference on Innovation and technology in computer science education Pages 25-28.

[3] Tamarisk Scholtz, Ian Sanders. Mental Models of Recursion: Investigating Students' Understanding of Recursion. University of the Witwatersrand, 2010. Proceedings of the fifteenth annual conference on Innovation and technology in computer science education Pages 103-107.

[4] Alimucaj, A. (2009). Eclipse Control Flow Graph Generator. Retrieved January 31, 2019, from http://eclipsefcg.sourceforge.net/

[5] Code Visualization.
https://www.grammatech.com/products/code-visualization

[6] Code2Flow.
https://code2flow.com

[7] Halim, S. (2011). VisuAlgo.net/en. Retrieved February 4, 2019, from https://visualgo.net/en

[8] AlZoubi, Omar & Fossati, Davide & Di Eugenio, Barbara & Green, Nick & Alizadeh, Mehrdad & Harsley, Rachel. (2015). A Hybrid Model for Teaching Recursion. The 16th ACM Annual Conference on Information Technology Education, Chicago, USA. 10.1145/2808006.2808030.

[9] JavaVisualizer.
https://cscircles.cemc.uwaterloo.ca/java_visualize/

[10] InfoVis Toolkit.
http://ivtk.sourceforge.net/

[11] LogicVis User Manual
https://github.com/orenjina/LogicVis/blob/master/User%20Manual.pdf