LogicVis Project Proposal

CSE 403 Software Engineering
Winter 2019

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

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. A method we found that is effective is to visualize code in the form of control structure diagram, graphs with logic flows that helps people understand codes, since it connects with the prior knowledge of aspiring students well[1]. Though graphic tools exist, we noticed a distinct trend amongst all the tools we checked, like JavaVisualizer, code2flow, and InfoVis Toolkit --they do not handle recursive cases. 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]):

1. 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. 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.

2. Lack of a proper methodology to represent a recursive solution
Many students that are new to recursion do not understand how to write a solution that uses recursion. They begin to write recursive solutions when prompted, but often can not imagine the solution they drafted in a logical way. For example, including a base case or a recursive call, but being unable to connect these components together to form a coherent solution.

The goal of our project is to build a tool for Java that can help programmers learn recursion by visualizing the flow behind programs into graphical forms. We want a tool that can translate a chunk of recursive code to a flowchart so that it is easier for people to grasp the recursive algorithm. Furthermore, we intend to build dynamic analysis tools to assist students with understanding their own code. The tool steps through the lines in the algorithms in the same way, so users can follow the reason behind each action it performs well.

The direct answer to why it would solve the two major issues students have above will be included in more details in later sections.

Current 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. However, the visualization it produces neither shows what each line of code is doing (each lines 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, it and many other tools[5][6] falls short in that it does not handle recursion any more than simply noting that the method was called. In the example of the Eclipse CFG Generator, it 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, it does not give much assistance in understanding recursion. It helps little for a student trying to learn recursion by looking at such visualization because it visualizes nothing more than one function is being called.

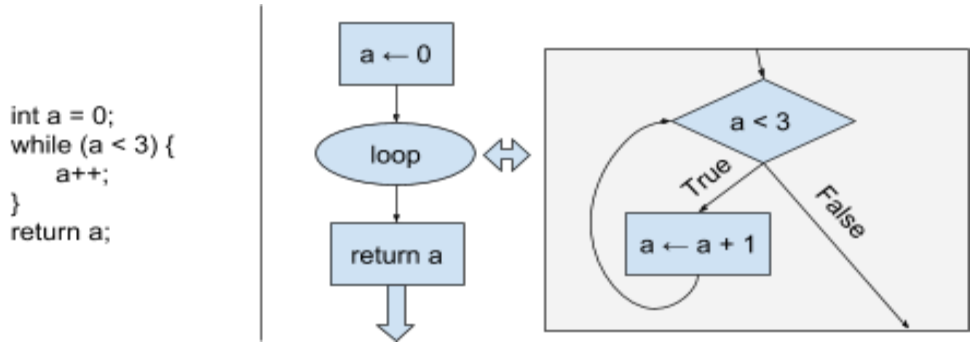
The best recursion trackers we found online were those like VisuAlgo[7]. VisuAlgo runs a program, such as GCD, and demonstrates a live example of the method call flow as a recursive tree. However, it only tracks the parameters and return value, making it difficult for students who are trying to understand this particular recursive function because the are not are of what is happening inside the recursive function as this detailed part of code is not visualized. It is also useless for programs that print out information in the middle of the function call, like print out a tree using left first depth first traversal, for the same reason, where students have no idea what happened before each print statement. Furthermore, the contents of the recursive method are never seen in the generated tree, making it difficult to understand why each recursive call was made (since it only shows that the call was made). We have looked for many tools that track more complicated recursive calls, but no visualizations exist.

Another approach on helping student learning recursion is ChiQat-Tutor System, a system that helps visualize certain predefined recursive functions[8]. It is helpful to get students to understand the given recursive function cases, but it fails to generalize on all recursive functions. Since this tool is not able to visualize all recursive calls, student may still find it difficult to reason about the own recursive code during debugging process.

Our Approach

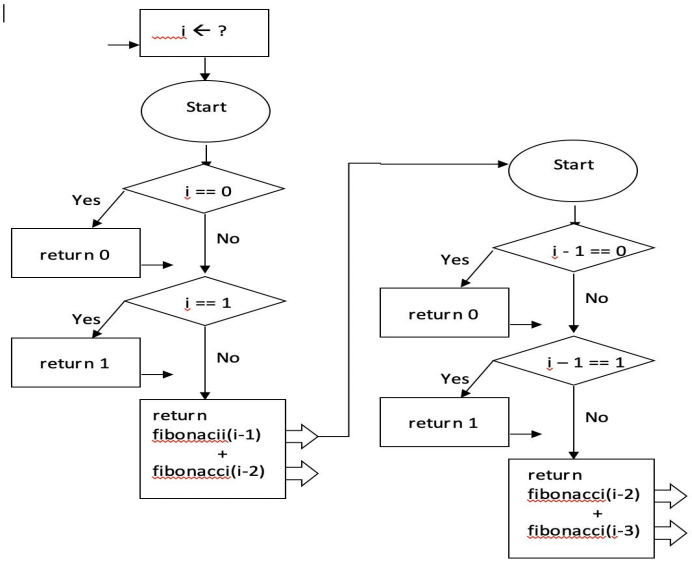
LogicVis is a program that takes an input of a piece of Java function code, primarily recursive functions, and outputs a logic graph of control flow. It reads in the Java code line by line and turns each line into a node representation with the shape indicating the type of command. LogicVis graphs contain different representations for basic Computer Science concepts, including if/else, loops, recursion, etc. We are implementing a method that takes in a piece of Java code containing a recursive function and outputs a built graph object that contains the logic

visualization with simple variable value tracking for the function as the initial product. The following figure indicates the simple design of the graph with no recursion.



To deal with the problem that students sometimes fail to recognize the recursive function they call as an abstraction, we are building our graph to represent those recursive steps concretely. Our take on the visualization of a recursive function will exhaustively list iterations of recursive calls. If the students can conveniently see the results by expanding the graphs, they have the potential to acknowledge and trust that the recursive call will perform its intended function even when the students haven't completed the program yet.

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. Though current debuggers and other tools like Eclipse provide instruction pointer tracing and variable tracking, students are still confused by many aspects of recursion. Research has shown that, many students forget or simply does not know that the current function is suspended once it calls the subsequent recursive function. As one result of such mistake, they use the wrong variable values when analyzing those functions [8]. In the graph we build, we will have the similar functions such as the current instruction pointer and variable values when run, but we will also have made it clear that another function frame has been made every time we descend down the stack.



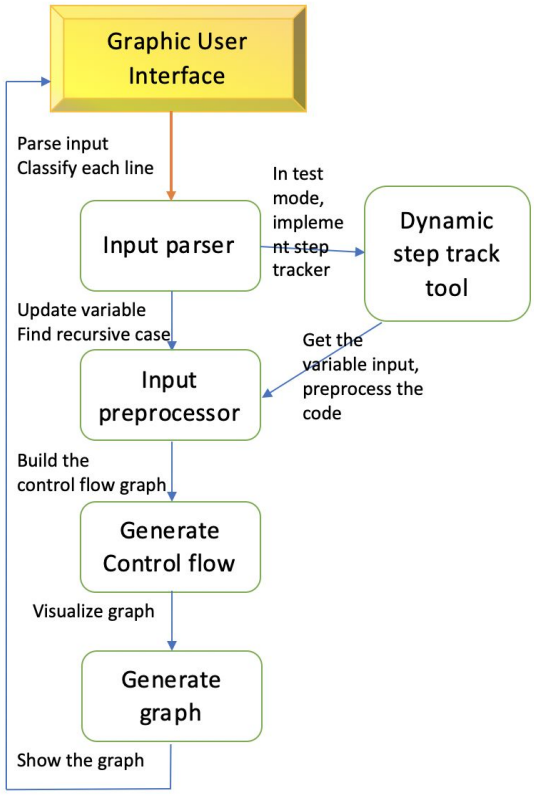
The recursion will be handled by adding options to expand recursive calls when they appear in the original function graph. They will be generated on demand and allowed a complete expansion in concrete cases. Our minimum viable product will be designed such that a strict number of recursion iterations 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.

If we have time in the end, we want to potentially look at implement this product as a plugin to Eclipse or supporting modification on the graph generated being reflected in the code. Since we are not planning on basing our product off of any existing products or purchasing outside resource, our cost plan, in this case, is 0. We have allocated 8 weeks based on our designed functionality of the final product. However, we did also account for estimation errors by designing a Minimum Viable Product that will be implemented first and a stretch goal in case we have time.

Implementation

Architecture

For this project, for the static recursive function logic graph generator, we decided to implement with two layers: the Graphic User Interface Layer, and the Graph Generator Algorithm Layer. The Graphic User Interface is first displayed to the user with the option of inputting code, as the users passing in the code to be visualized and the values they want to test out. Then it goes to input parser which will parse the input (recursive function), it will display error if the code is not valid Java code. If it is the static mode the input preprocessor will preprocess input code in the static rule, if it is dynamic mode, it will implement a dynamic step track tool which allows user to execute code step by step, then the preprocessor will preprocess input code in the dynamic rule. After preprocessing, control flow generator will process the logic and generate the control flow, then the graph generator will generate a graphic output based on the control flow. Lastly, the graph will be displayed in the GUI.



Technologies

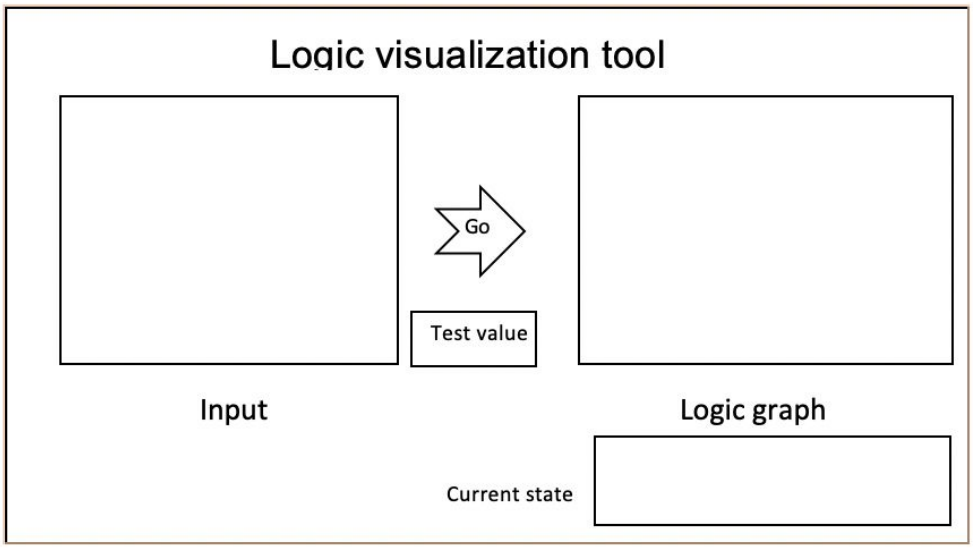
We have chosen to develop this software in Java since we are most familiar with the Java language and its libraries.

Graphic User Interface:

We plan to 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.
Graph Generator Algorithm:
We plan to build this from scratch. We want to be able to make a parsing and graph generation software that fits our Graphic User Interface, so although we may reference existing flowchart generators, we would have to rewrite most of it if we wanted to use one.

Interfaces

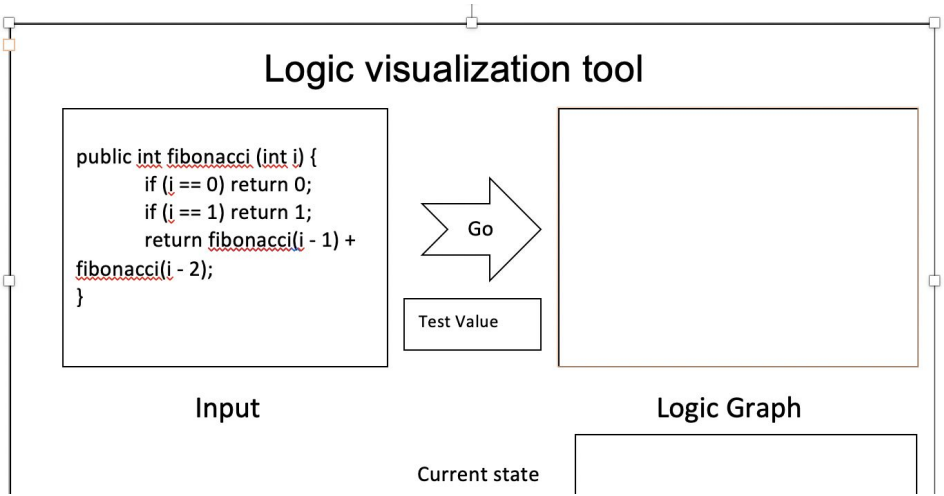
Graphical User Interface prototype:



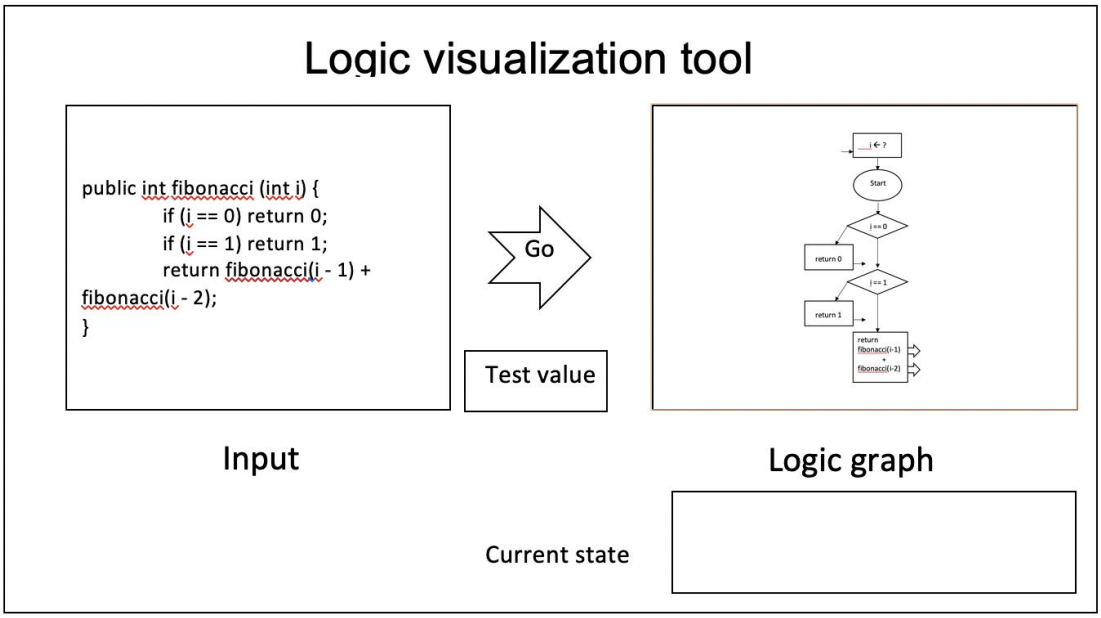
It consists of an input box in which users can type the recursive function that they want to translate, a Go button which will do the translation, a "Test value" button which will change to the test mode, a current state box which will show the variable values in the test mode and output box which will show the graph.

In the input box, users are able to type the input code. For example:

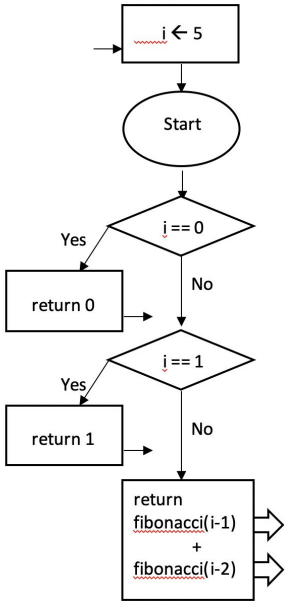
```
public int fibonacci (int i) {
        if (i == 0) return 0;
        if (i == 1) return 1;
        return fibonacci(i - 1) + fibonacci(i - 2);
}
```



After pressing the "Go" button the Logic graph box will show the logic graph where the parameter i is an indefinite value:

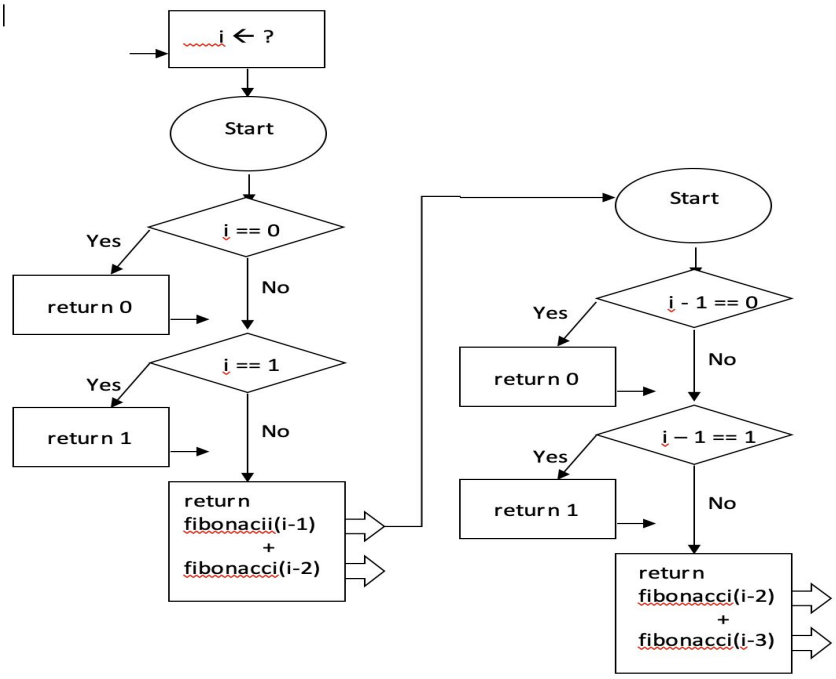


In this graph, both arrows in the last box can be clicked because they are calling itself in recursion.



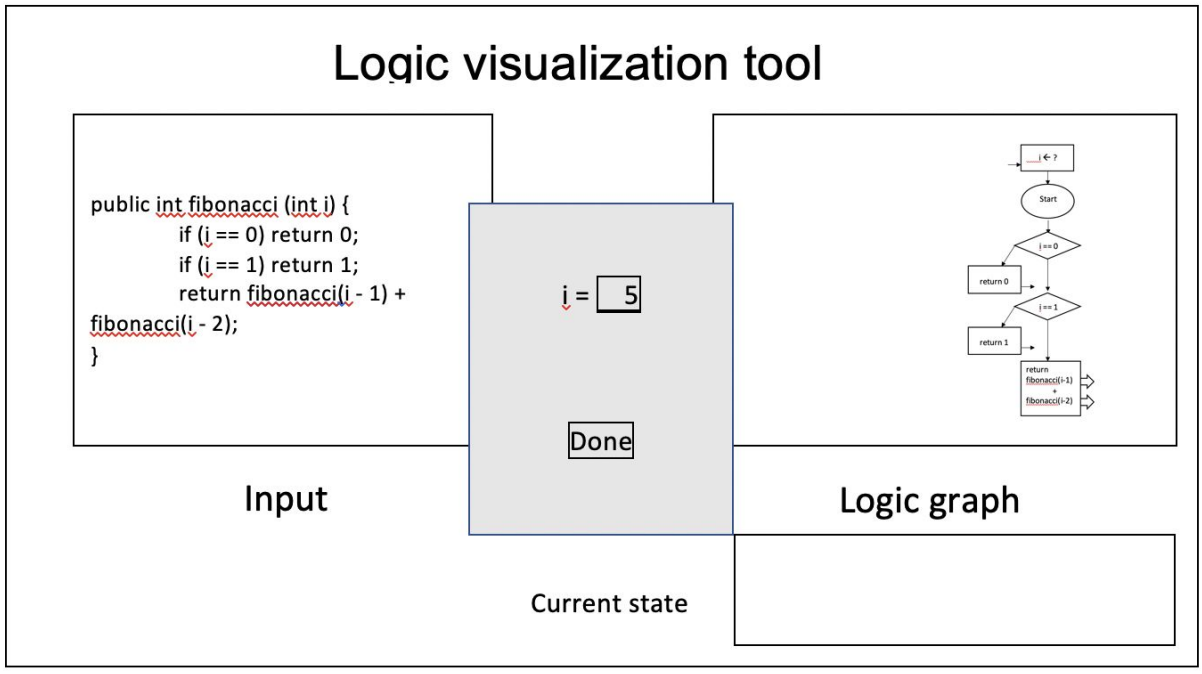
(Arrows are clickable)

After clicking the arrow for fibonacci(i - 1), in the output box we get:

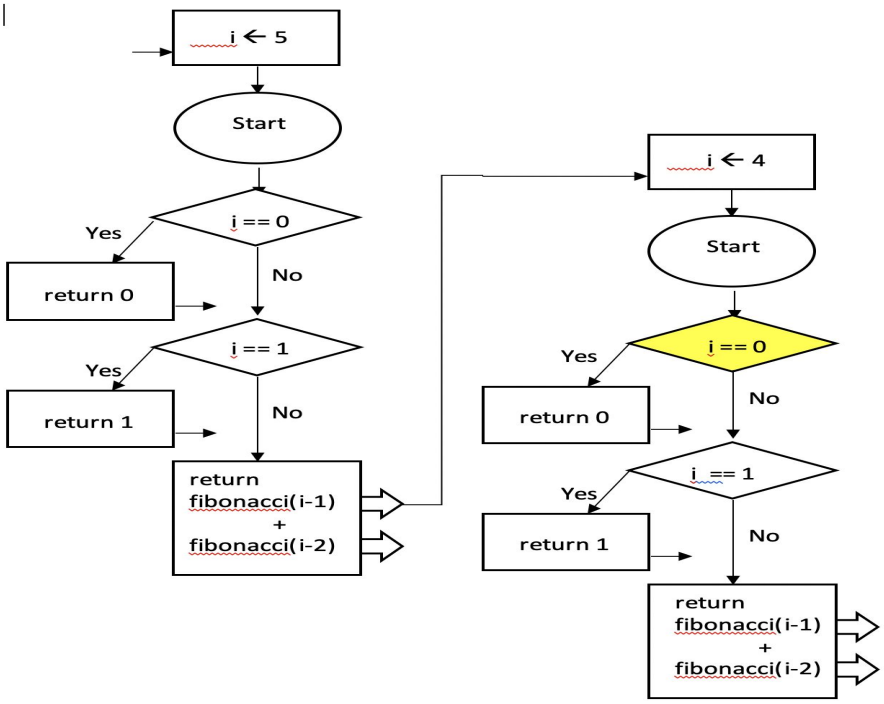


This shows an extended graph with a recursive function and in the recursive function, i becomes i - 1.

In the main panel, we have a test value button. After clicking it, it will ask for the input value for i.

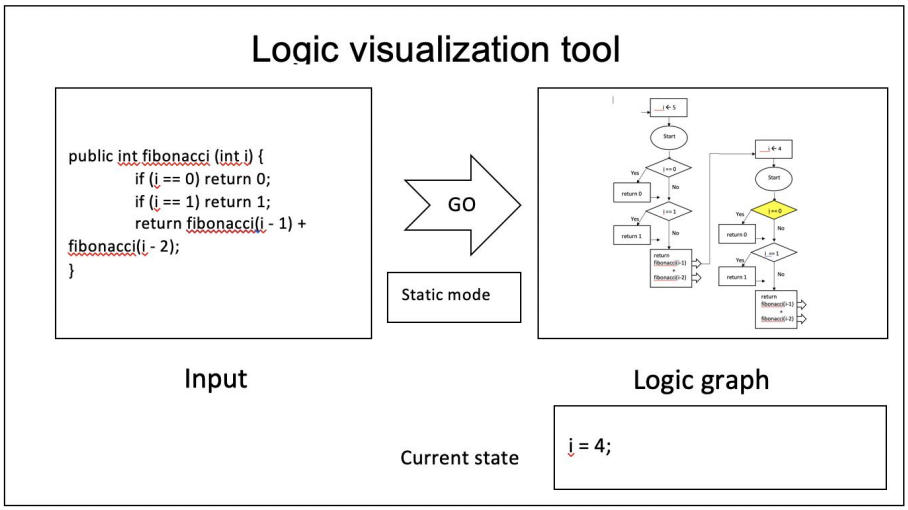


If we typed i = 5, the graph will be changed to test mode (dynamic mode). In the test mode, just like a debugging tool, it can ask users to step forward. User can click the image output block to execute next step. the block with yellow color will show the current step:

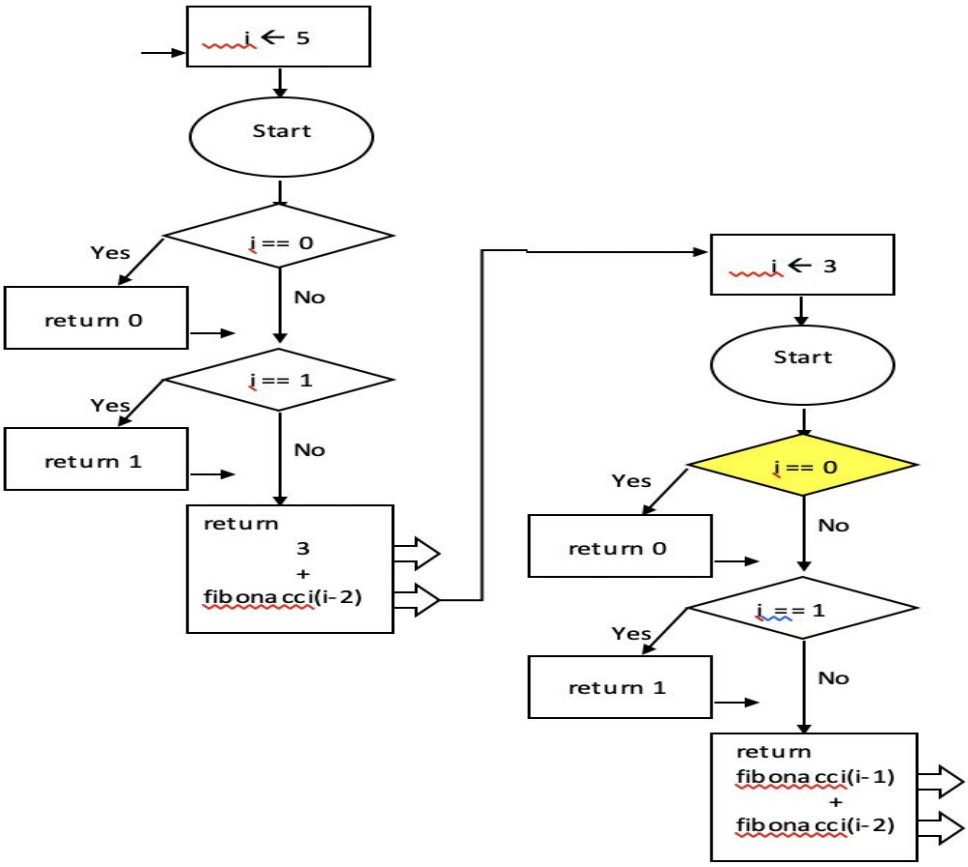


Other several differences between the test mode and indefinite value mode are:

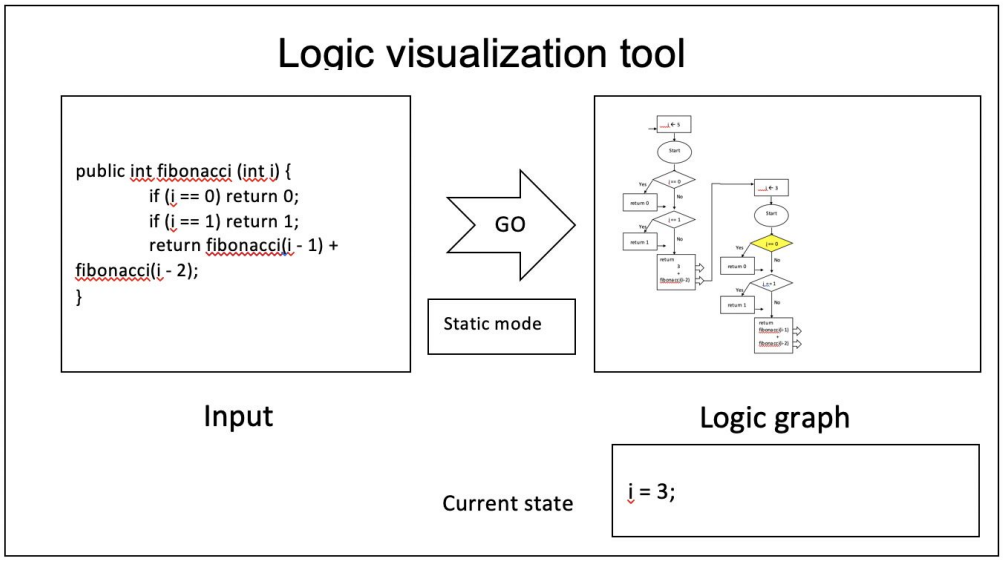
1. The recursive function call will have a value for i before the start block
2. Instead of showing i - 1 like the previous graph, it will show i instead with i has value 4.
3. In the main panel, the current state block will show the variable values for the current state. In the example above, when we are at the yellow step, the current state block will show i = 4



We move forward to the second recursive function fibonacci(i-2):



(4) After finishing the first recursive function fibonacci(i-1), it will change the function to a returned value 3. In the example above, the current state block will show i = 3.



Limits and Risks

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 harmful impact of such risks, we plan on conducting user research after the product has been built as well as having researched using paper-prototyping already.

We also have taken risks in deciding the scope of our project. We have chosen to make this software specifically for Java, which means that building a graph for any other language is not possible. We based this on the idea that most students that learn computer science start with Java. 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. So, we currently have chosen to treat imported functions like a generic line, but we may change this depending on the feedback we receive when doing research. 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.

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	- Implement improvements on the graphs if found necessary by testing - Build the frame of the GUI
Week 9	- Finish the GUI - Research on prefered GUI components using testers
Week 10	- Handle multiple functions - Link relations between existing functions
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. 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.

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

User Testing

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

private static void factors(int x, int n) {
    if (n == x) {  // base case
        System.out.print(x);  // biggest factor; we are done!
    } else if (x % n == 0) {  // we have found a factor
        factors(x / n, n);  // continue with dividend
        System.out.print(n);  // print it
    } else {  // not a factor
        factors(x, n + 1);  // check next possible factor
    }
}
```

This is the code for user testing. The code is recursive and somewhat uncommon so users will not know the process without understanding the code. We have developed a

paper-prototype and tested the effectiveness of it using this piece of code. The paper prototype replicates the GUI as defined above.
Result:
Prompt: What does this do? What is the output of mystery(20)?
Response: factors of whatever you put in based off answer output: checks if divisible by 2, then by odd prime(?). 2 2 5.

Prompt: Try again with mystery(30)
2 3 5. returns factors from smallest to greatest; checks 2, then 3.

Prompt: Try again with mystery(22)
2 11. Has good grasp of how this code works; checks numbers starting from 2 upwards

User Feedback:
- n > x / n is too much math: Our chart may not help people understand why that's there
- Flowcharts make it easier to see steps: Faster math/processing
- Confirmed: flowchart representation helps you see Recursion
    - shows how it goes on to the next one
    - can see what's happening, especially when not as used to code
    - flowcharts are universal (not as important)
- Output is very nice
- Primarily helps with understand control flow (again, for beginners)
    - if/else, loops, and such would be necessary

User Suggestion: 3 types of visualization
- "Verbose" is default
    - Verbose is what our current flowchart looks like
- "Simple" is a bit harder to remember steps
    - Simple took out all the "irrelevant steps"
- "Brief" would be useful for finding answers
    - Brief only includes prints, returns, and recursive calls: only the "impactful" things

2nd test - Fibonacci Sequence
Test: the Fibonacci sequence program without the paper prototype
Observations:
- Our tool is good at doing math quickly
- Had to write down own way of understanding code flow
- In most cases, helps keep track of where you are

Things we will test:
- more people
- operations after recursive call
- modifying block flow + impact (aka the reverse question)

return values
bugged code (e.g. infinite loops)

The user tested has a background of having learned recursion, but not well and did not learn further computer science for a few years. The user feedback given by the user study reflected that our prototype does help track where the code is at in terms of recursive iteration and current instruction. This helps solve the problems many have with recursion, such as not comprehending the what the recursive call does, which first reason given in the motivation section. In addition, the user also showed a need to write the user's own interpretation of the program during the Fibonnaci sequence. In comparison with the first program where the user did not need to do this, we draw the conclusion that the assist our program provides can be used as a sort of representation of recursion, which provides a solution to the second problem users have: a lack of representation when drafting a recursive solution.

Further testing may be required, but the initial paper-prototype testing supports our reasoning that the product will make recursion easier to manage for learning students.

Citations

[1] Cross II, J.H. & Sheppard, S.V.. (1988). The control structure diagram. 274 - 278. 10.1109/PCCC.1988.10084.

[2] Raja Sooriamurthi. Problems in Comprehending Recursion and Suggested Solutions. University of West Florida, 2001.

[3] Tamarisk Scholtz, Ian Sanders. Mental Models of Recursion: Investigating Students' Understanding of Recursion. University of the Witwatersrand, 2010.

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

[5] Grammatech's tool is not free and focuses on understanding programs at a larger scale (i.e. multiple classes and function calls between them).
Rfleming. (2015, December 30). Code Visualization. Retrieved January 29, 2019, from https://www.grammatech.com/products/code-visualization

[6] Code2Flow is an online code visualization tool that suffers from the same issue as the Eclipse CFG Generator in that it does not handle recursion in a clear way.
The simplest way to describe your flows. (n.d.). Retrieved February 3, 2019, from 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. 10.1145/2808006.2808030.