

LogicVis User Manual

Software Description

LogicVis is a program that takes an input of a piece of Java function code, primarily recursive functions, and outputs a visual representation of that code.

Description

LogicVis is fundamentally a program that helps people understanding the code and particularly recursion. It takes Java code and transforms it into an easy-to-understand flowchart that reflects the logic of the code. It comes with a user-friendly interface with simple and intuitive operations so that a person with any level of technical background is able to use it. This program is completely free to use.

Benefits and Values

- Our program generates flowcharts with easy to recognize shapes representing each conditional statements and loops.
- Our program provides exhaustive visualization of the recursive calls. (assuming the recursion is not infinite)
- Our program enables progress tracing with current instruction pointer, variable value indicator, and function call stack frames.

System Requirements

Before using this software, the following must be installed:

- Java 8

Building from Source

Required software: Maven, JDK 11 (JDK 11 is only required to build from source. Java 8 suffices if only trying to run the jar executable.)

Run “mvn clean install” from the terminal in the project repository.

To launch the tool afterwards, run “mvn exec:java”.

Using the System

This software is able to translate a Java method into a corresponding flowchart that has the same meaning. From there, it can track variables throughout the method and recursive method calls, if any.

Please do not use this tool before ensuring that the method has no compiler errors.

Understanding the Interface

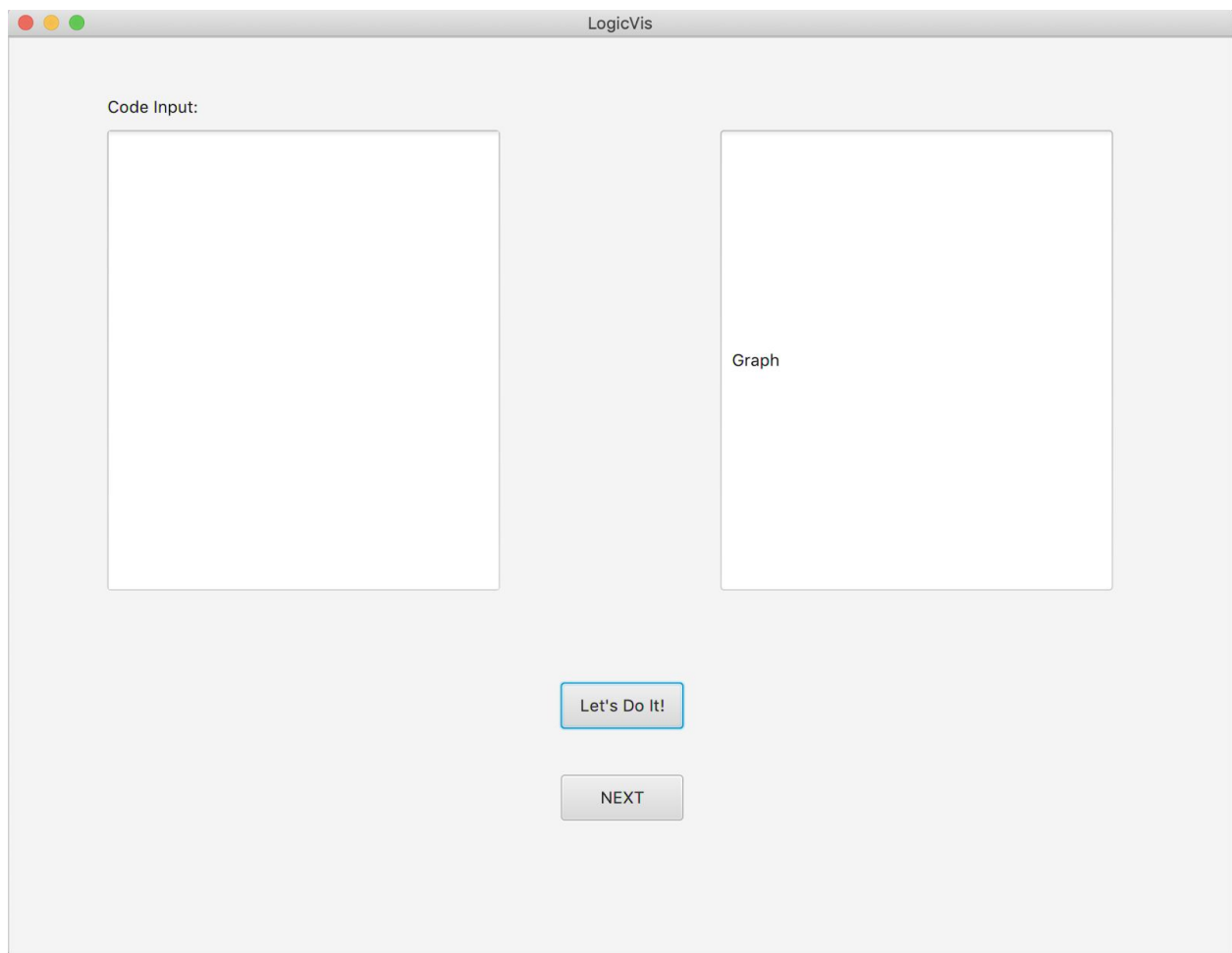


Fig. 1 - LogicVis tool as seen on startup


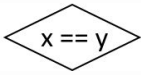
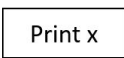
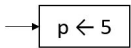
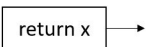
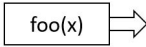
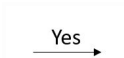

The Logic Visualization tool interface consists of a few parts:

- **“Code Input” box** in which users can type the recursive function that they want to translate
- **“Let’s Do It!”** button which will trigger a pop-up dialog box if there is a valid code input in the code input box.

- **“Next”** button will step the recursive process one at a time.
- **Output box** on the right that will display a flowchart

Reading the flowchart

In the output flowchart, there are shapes and arrows with different meanings.

Name	Figure	Description
Ellipse		Denotes the entry into the method as well as the beginning of a cycle, such as a loop or recursive call.
Diamond		Denotes a branch in the flowchart, such as an if statement or switch-case statement. Multiple conditioned arrows will branch from this shape.
Rectangle		Used to represent any piece of code that does not uniquely affect the control flow of the program. This includes variable setting, return values, and many others.
Arrow Inwards		Denotes a parameter, as written inside the box. These are only found before Start.
Arrow Outwards		Denotes a return value, as written inside the box.
Big Arrow		Indicates a recursive method call. After putting in a test value, clicking the arrow will expand the method call.
Conditioned Arrow		Denotes the next step, if the previous statement matches the condition. Always originates from a diamond shape. <ul style="list-style-type: none"> - Yes means the previous condition was true, and No means it was false
Unconditioned Arrow		Denotes the following step.

Generating a Flowchart from a Java Method

This section details the process to take a Java method, translating it into a flowchart, and how to read the flow chart.

Generating the Initial Flowchart

1. Copy recursive function into the input box

Copy your method into the “Code Input” box of the tool, as seen in Fig. 2.

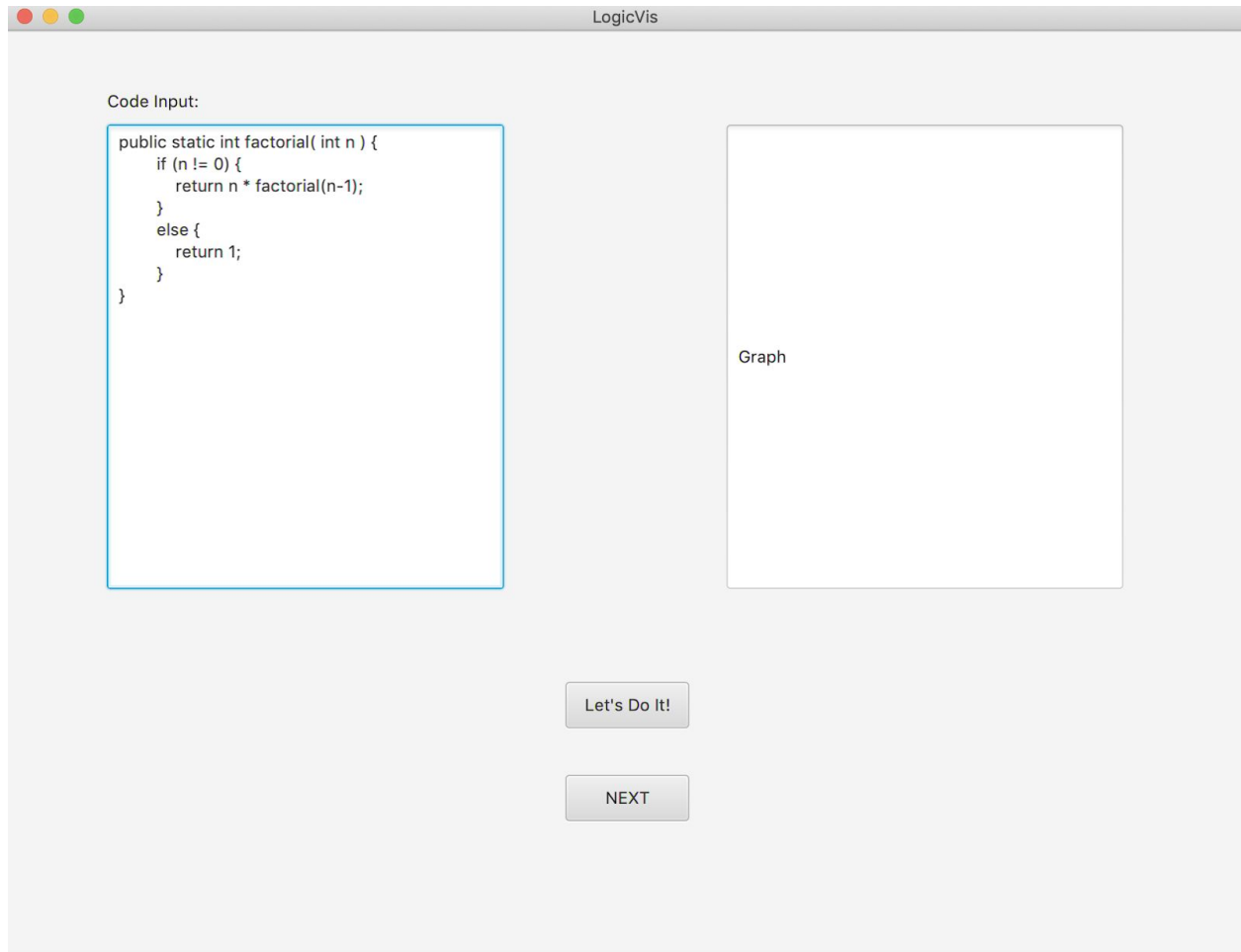


Fig. 2 - Write your function in the input

2. Press “Let’s Do it!”

If there are no errors in the input function, it will generate a dialog like in Fig. 3.

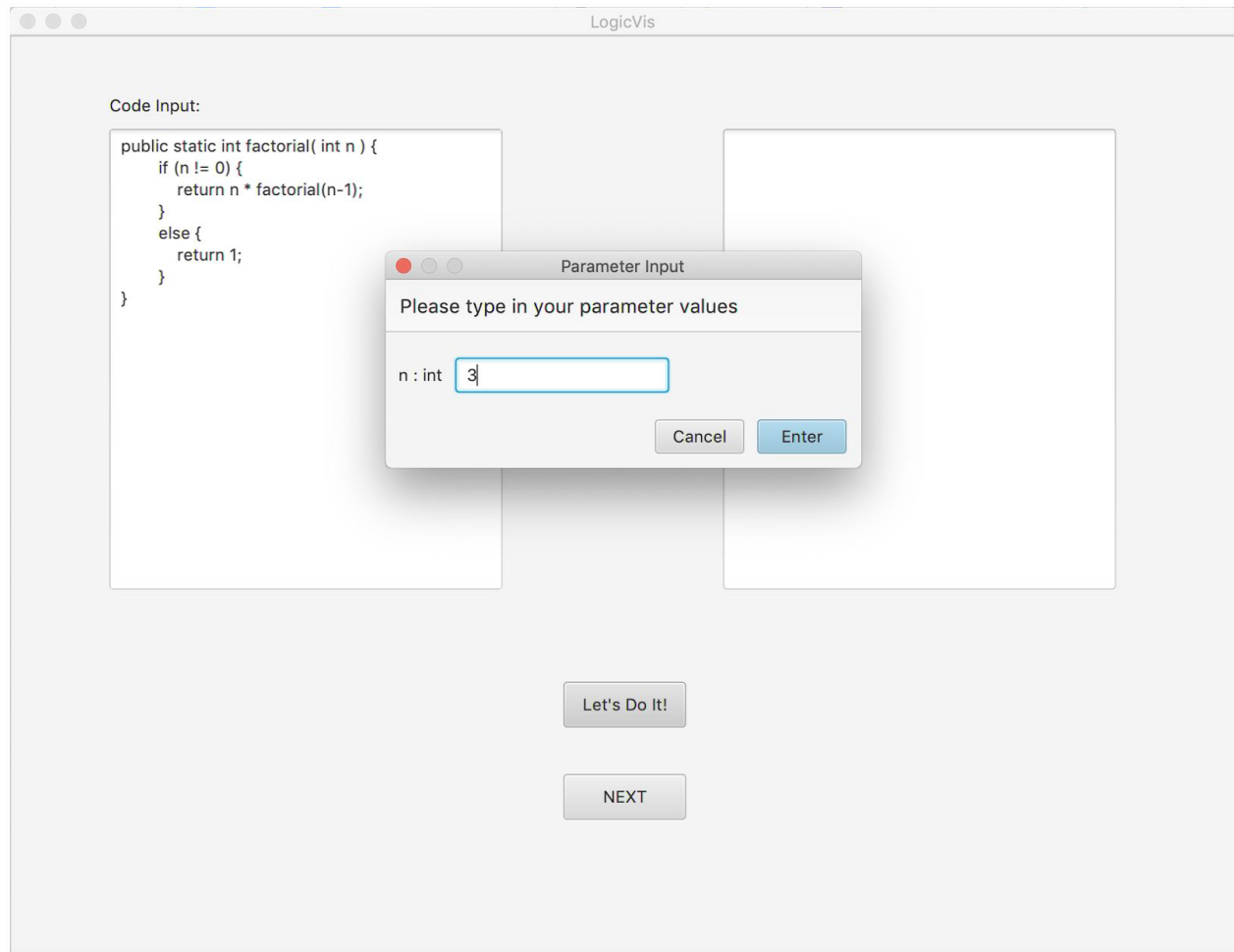


Fig. 3 - Input your parameter

3. Enter your parameters

Type in your parameters in the boxes indicated. Then press “Enter” to generate the graph, otherwise press “Cancel” if you are not ready. Once you “Enter”, the graph will generate like Fig. 4. It is possible to zoom in on the graph by clicking on the graph once. Click on the graph again to undo that. The return value of this will be indicated below the graph.

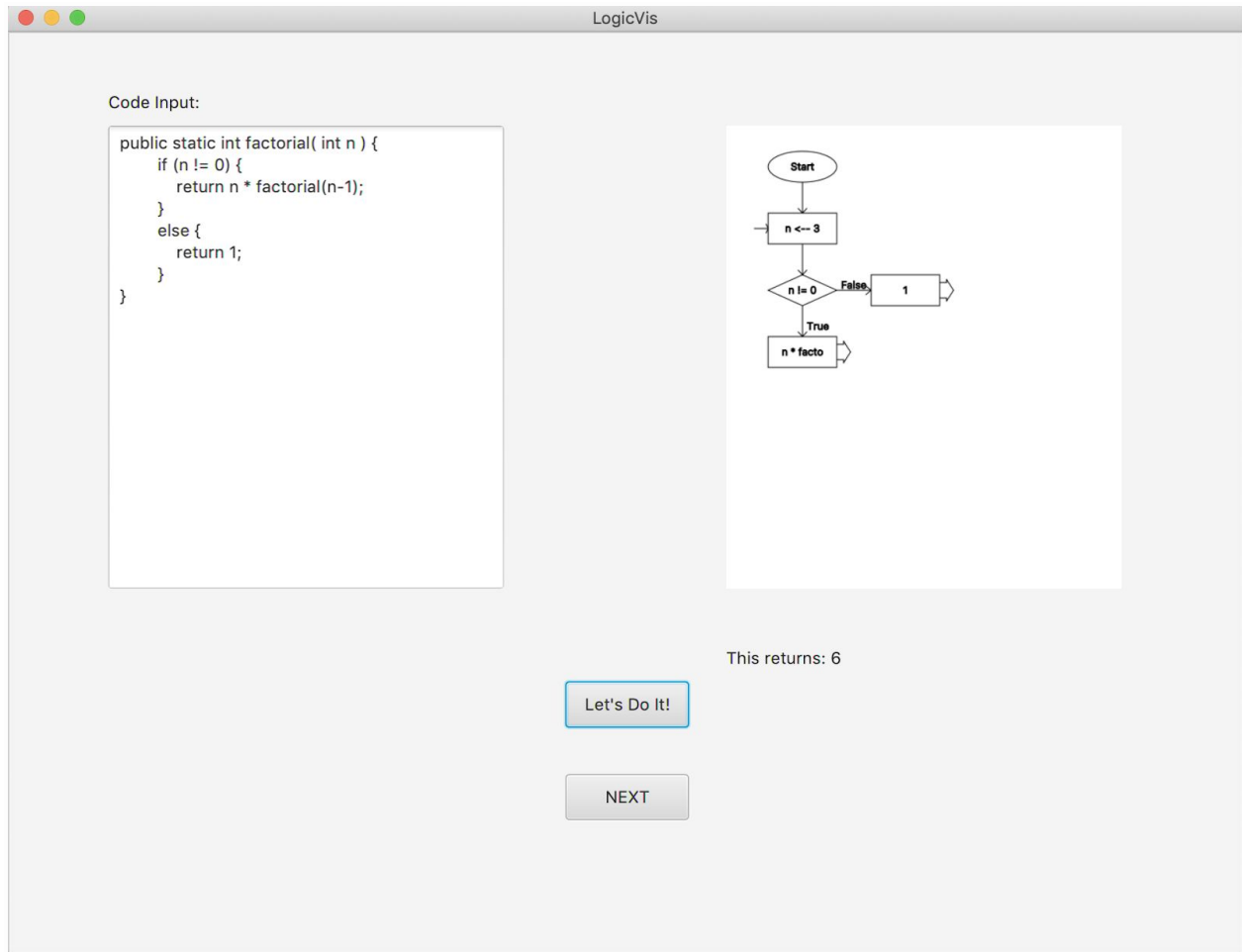


Fig. 4 - Base graph generation

4. To expand recursive calls, click “Next”

It is possible to step into a recursive call within the recursive method. This is done simply by clicking the “**Next**” button

For example, Fig. 5 shows the output section of the software. After **clicking “Next”** the graph will show the recursive function in an extended graph. It is possible to scroll horizontally to see the expansion of the graphs. The exact call that creates the next graph will be highlighted in **red**.

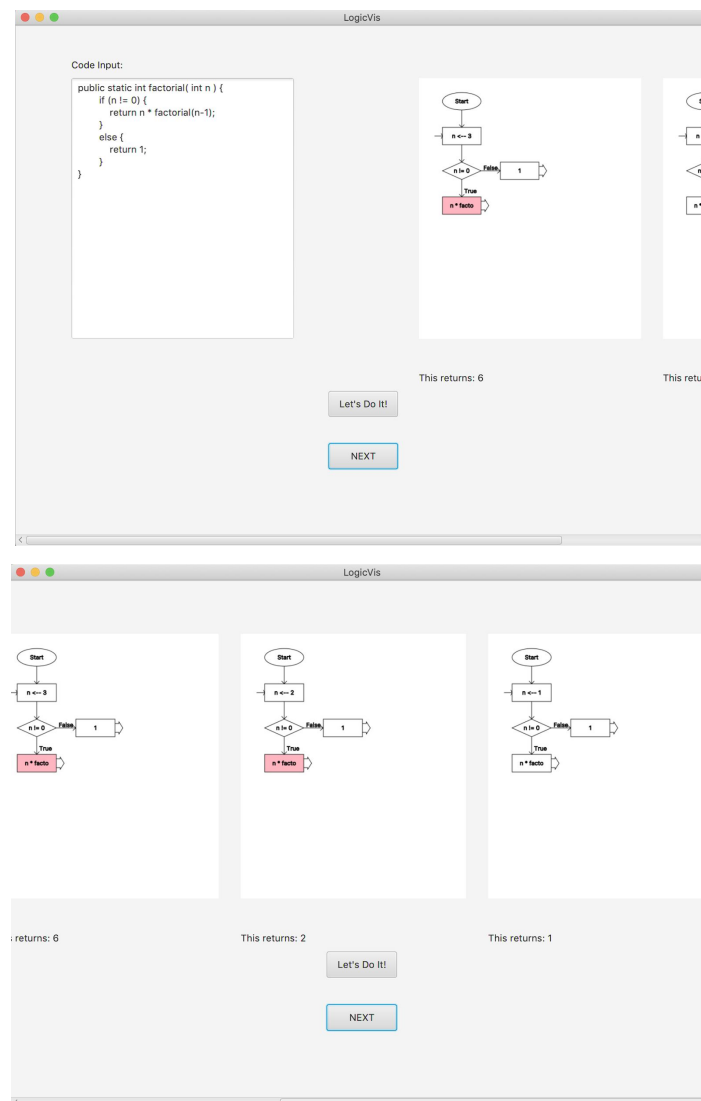


Fig. 5 - the output graph generated by a single next (up), and the output graph generated by 2 “next”s and scrolled to the right (down)

5. Keep clicking “Next” will keep extending the graph

Every time clicking “**Next**” will extend the graph with the next function call. When the graph hits the based case and starts returning, it will start shrinking and filling in the return values of the cases before until it hits the original case.

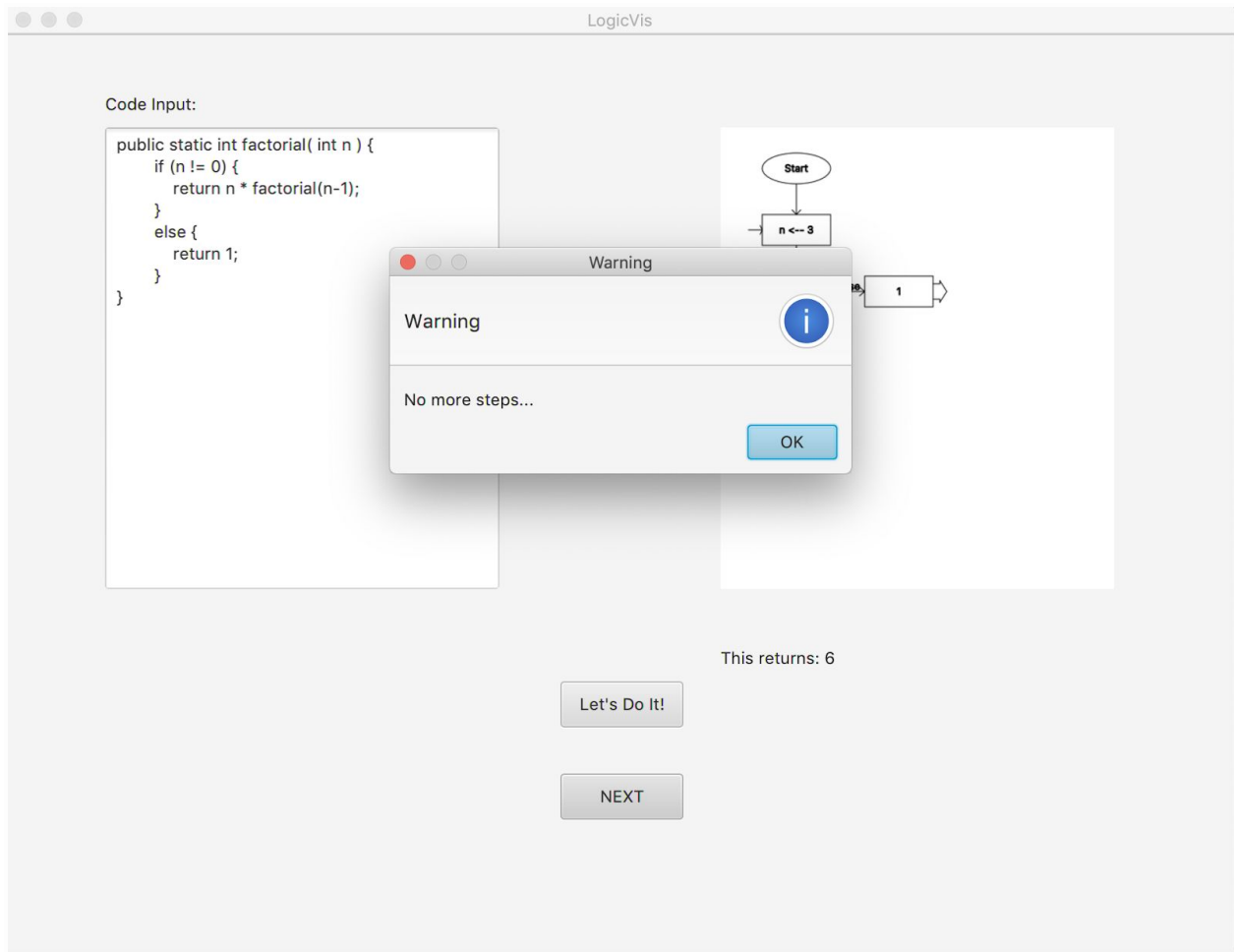


Fig. 6 - the output graph after finishing all steps and called next once again

Input Restrictions

This section notes a few input restrictions of the program. If these restrictions are violated, the program behaves unpredictably:

- Do not use any of these variable names in our input method: ROOT, curDEPTH, depTH, callFromLAST, returnVALUE
- Do not put any uncompileable code in our input method
- Do leave a new line at the end of a program
- Do run “mvn clean” and build again after an error if everything starts to crash

Bug List

This list presents known bugs that have not been fixed yet:

- 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.
- In 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;}”
- Sometimes the program would be unable to show the result of the first program.