Fibonacci Sequence:

Implement the Fibonacci function in both a recursive and iterative fashion. What's the runtime efficiency of each?

- **Turn in a chart of the results**, with time on the Y axis, and input on the X axis, Please use nanosecond. long startTime = System.nanoTime(); This chart must not be hand written.

Solution

To solve this problem, a fibonacci function got implemented via Recursion and Iteration, their execution times were measured and then compared in a plot. The **JMathPlot repo** was used to create the plot.

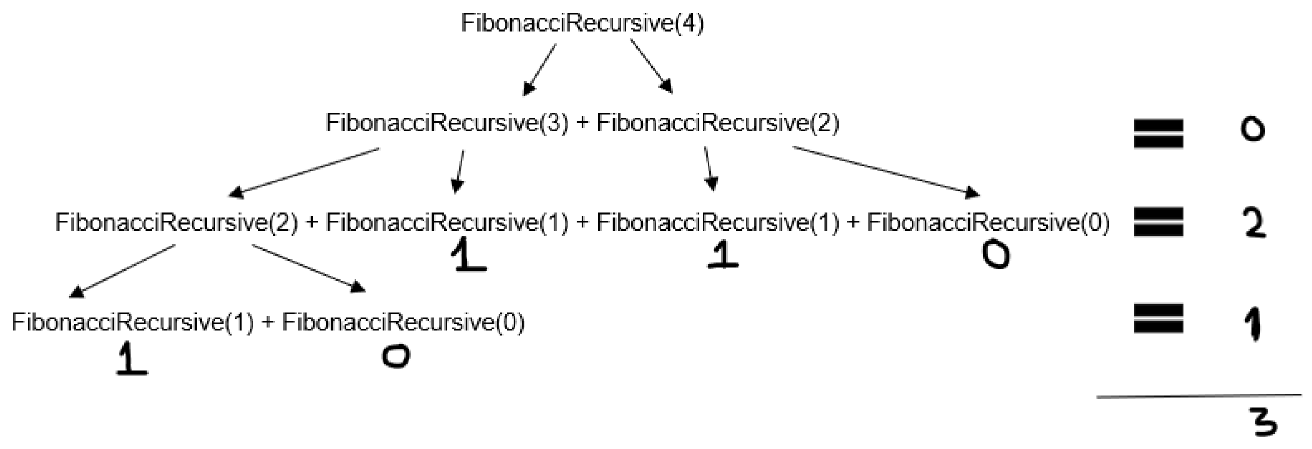
Design Walkthrough

To set up the environment, add the jars in the jar folder to your Java Classpath, these two jars are part of Yann Richet's **JMathPlot repo** which is used to plot the arrays after the program is done.

Here is the Recursive Function used to calculate the fibonacci sequence and the execution time: The recursive function, checks if the number is either 0 or 1 and returns it, otherwise, it returns the sum of the Fibonacci function of the variable - 1 plus the Fibonacci function of the variable - 2, and it continues to repeat itself until all functions return a 0 or a 1.

```
public static long FibonacciRecursive(long n) {
    if (n == 0){ return n;}
    if (n == 1){ return n;}
        else
    return FibonacciRecursive(n - 1) + FibonacciRecursive(n - 2);
}
```

Esentially it does this:


FibonacciRecursive(4)
FibonacciRecursive(3) + FibonacciRecursive(2)
FibonacciRecursive(2) + FibonacciRecursive(1) + FibonacciRecursive(1) + FibonacciRecursive(0)
1
1
0
FibonacciRecursive(1) + FibonacciRecursive(0)
1
0
= 0
= 2
= 1
3

And its Iterative counterpart:

```
public static long FibonacciIterative(long n) {
int PrevPrev, PrevNumber = 0, CurrNumber = 1;
for (int i = 1; i < n ; i++) {
    PrevPrev = PrevNumber;
    PrevNumber = CurrNumber;
    CurrNumber = PrevPrev + PrevNumber;
    }
    return CurrNumber;
}
```

When it comes to making a graph in Java, we'll use **JMathPlot repo**, which is similar to Matplotlib in Python, and allows us to graph arrays utilizing JFrame and plots.

Creating the PlotPanel:

```
Plot2DPanel plot = new Plot2DPanel();
```

Defining the Legend Position and Title:

```
plot.addLegend("SOUTH");
BaseLabel title = new BaseLabel("Recursive vs Iterative Fibonacci:", Color.black,
0.5, 1.1);
title.setFont(new Font("Courier", Font.BOLD, 20));
plot.addPlotable(title);
```

Adding the line plots to the plot:

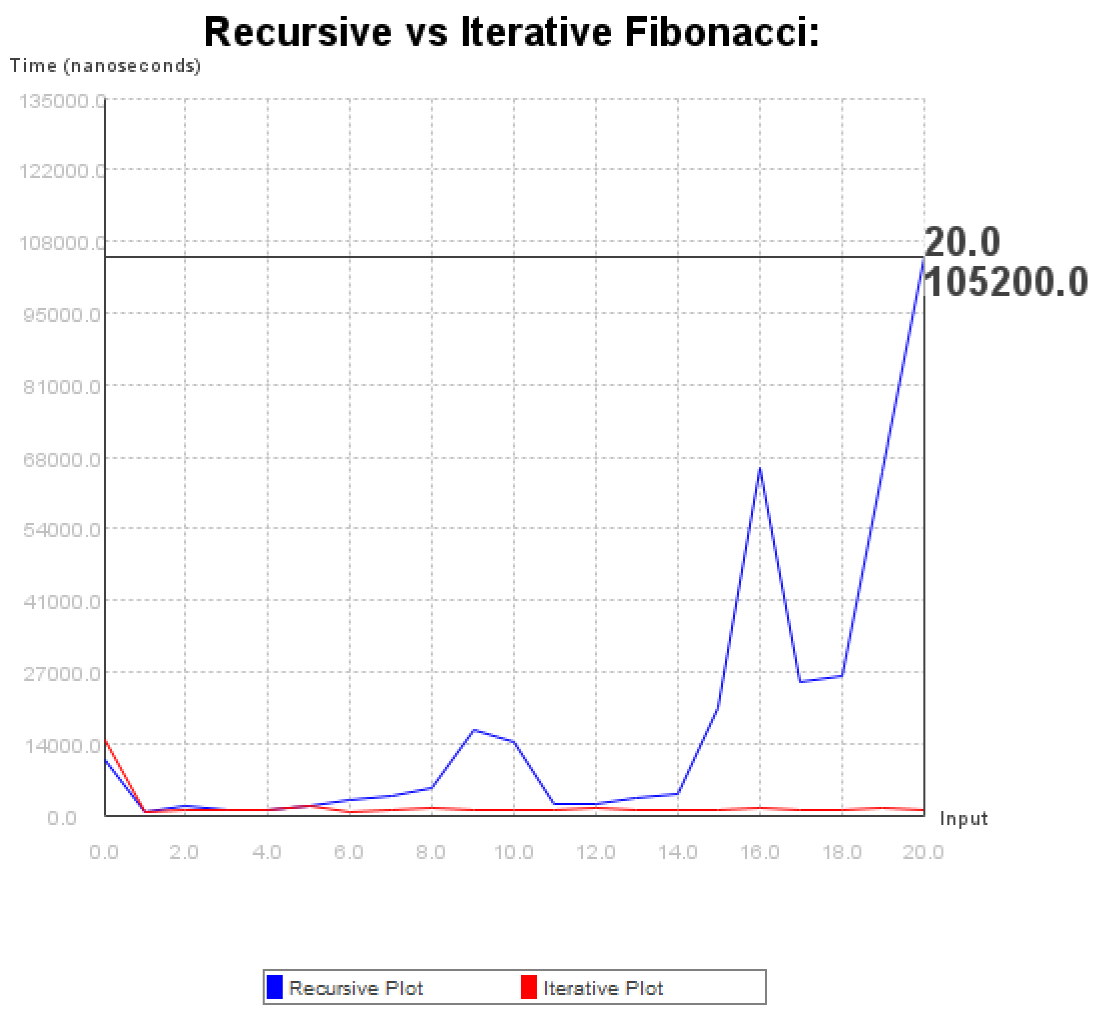
```
plot.addLinePlot("Recursive Plot", InputA, TimeA);
plot.addLinePlot("Iterative Plot", InputB, TimeB);
```

Finally, putting the Plot into a JFrame:

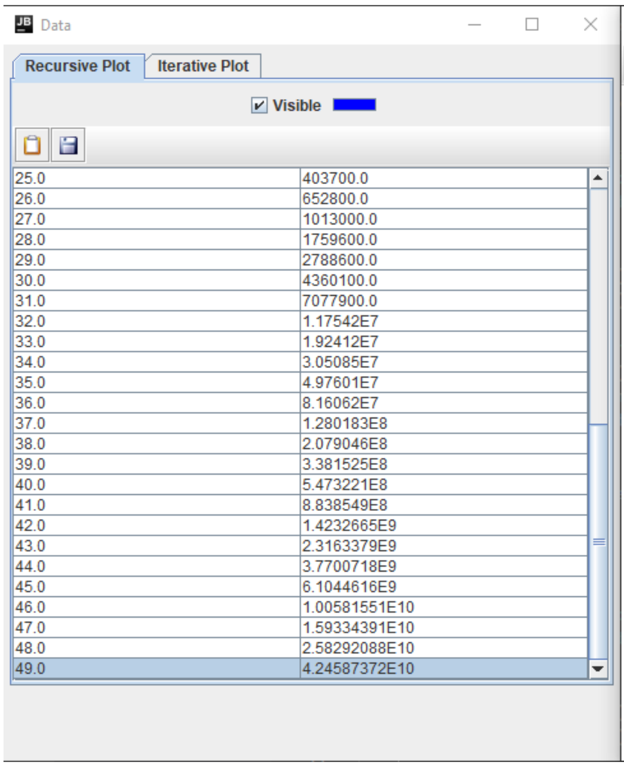
```
JFrame frame = new JFrame("Valencia College - Javier Silva");
frame.setSize(600, 600);
frame.setContentPane(plot);
frame.setVisible(true);
```

Output and Explanation

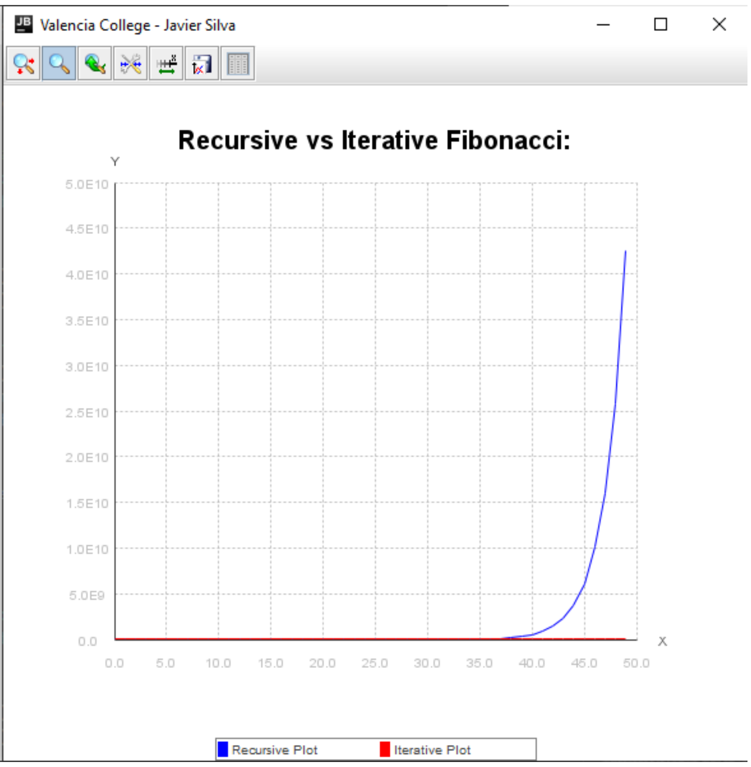
Here's the final graph of the execution times of the recursive function (blue) vs the iterative function (red):



We can clearly see how in this case, the recursive function takes longer to run than the iterative one, the reason why is that no matter the number, the iterative function will always be linear and run on O(1) time. Meanwhile, the recursive function grows at an exponential rate, or O(2^n), as it has to go through the entire tree of possible values before arriving at the solution. This exponential graph can be viewed perfectly when we graph the execution time of the function as we go from 0 to 49:



25.0	403700.0
26.0	652800.0
27.0	1013000.0
28.0	1759600.0
29.0	2788600.0
30.0	4360100.0
31.0	7077900.0
32.0	1.17542E7
33.0	1.92412E7
34.0	3.05085E7
35.0	4.97601E7
36.0	8.16062E7
37.0	1.280183E8
38.0	2.079046E8
39.0	3.381525E8
40.0	5.473221E8
41.0	8.838549E8
42.0	1.4232665E9
43.0	2.3163379E9
44.0	3.7700718E9
45.0	6.1044616E9
46.0	1.00581551E10
47.0	1.59334391E10
48.0	2.58292088E10
49.0	4.24587372E10


Valencia College - Javier Silva
Recursive vs Iterative Fibonacci:
Y
5.0E10
4.5E10
4.0E10
3.5E10
3.0E10
2.5E10
2.0E10
1.5E10
1.0E10
5.0E9
X
Recursive Plot
Iterative Plot