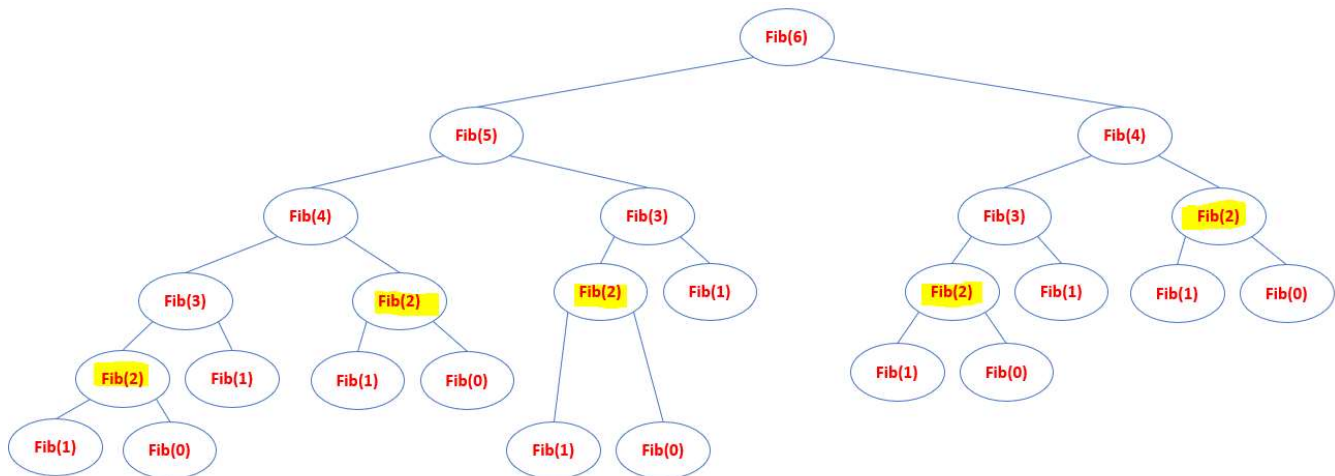


## Module 5 Programming Assignment Report

**Note to the TA:** This report exceeds the one-page limit due to the amount of information we are required to provide and answer in this module's programming assignment.

## I. Programming

- Using the naïve recursive (top-down) algorithm to compute Fibonacci(6), the number  $n_2$  times **Fibonacci(2)** was called is **5**; This is illustrated by the following recursive tree. We can see that **Fibonacci(2)**, highlighted in yellow, was called **5 times**.



- Here is the proposed pseudocode for the recursive version **with memoization**:

### INITIALIZE\_Fib-Array(n)

```

1  let Fib[1.. n] be a new array // In this programming assignment, the Fib array is declared as a global
2  for j = 1 to n                // variable so that the Fibonacci_Memoization(n) method can have
3      Fib[j] = 4                 // direct access to it. This reduces the execution time.

```

### Fibonacci\_Memoization(n)

```

1  if n ≤ 1
2      return n
3  if Fib[n] != 4
4      return Fib[n]
5  Fib[n] = Fibonacci_Memoization(n - 1) + Fibonacci_Memoization(n - 2)
6  return Fib[n]

```

- Here is the proposed pseudocode for the iterative bottom-up version:

### Fibonacci\_BottomUp(n)

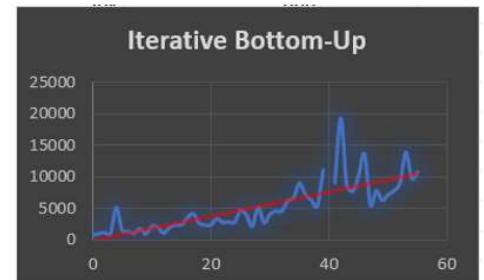
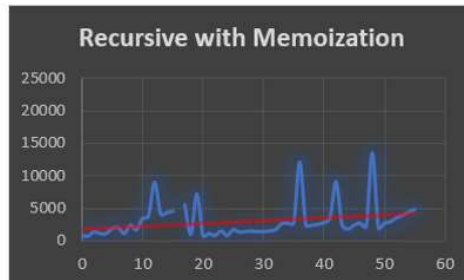
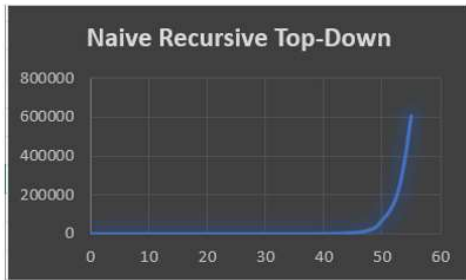
```

1  let F[1.. n + 2] be a new array
2  F[0] = 0;
3  F[1] = 1;
4  for i = 2 to n
5      F[i] = F[i - 1] + F[i - 2]
6  return F[n]

```

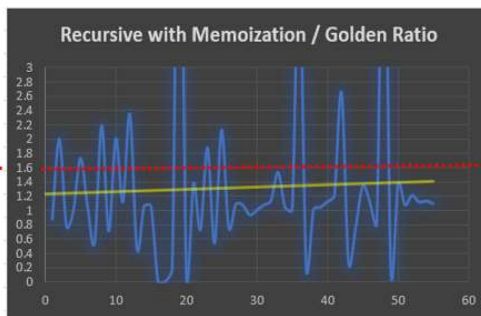
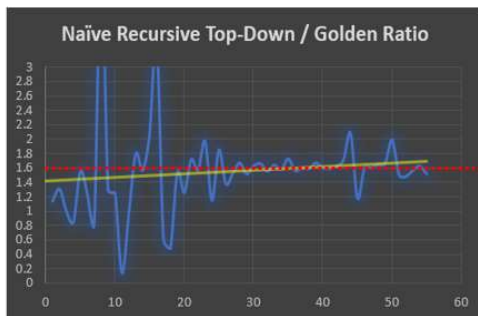
## II. Data Collection and Analysis

- The following are the plots of the execution times of the three different algorithms discussed in this programming assignment:



- The following are the plots of  $\frac{t_i}{t_i - 1}$  versus  $i$  for  $i = 1$  to  $i = 55$ :

----- Golden Ratio  
————— Trendline (~average)



We can see in the “**Naïve Recursive Top-Down**” execution time plot above that the function grows exponentially. Hence, we can say, based on the **Growth of Functions**, which we have learned in Algorithms I, that this function (i.e. algorithm) has a time complexity of  $O(2^n)$ . This is the slowest way to solve the Fibonacci Sequence because the algorithm computes exactly the same subproblems repeatedly. Thus, the amount of operations needed, for each level of recursion, grows exponentially as the depth approaches  $n$ .

We can also see in the “**Recursive with Memoization**” and “**Iterative Bottom-Up**” execution times plots that these two functions grow linearly. Again, applying what we have learned in the Growth of Functions in Algorithms I, we can say that both these functions (i.e. algorithms) have a time complexity of  $O(n)$ , i.e. each number, starting at 2 up to and including  $N$ , is visited, computed and then stored for access later on. While these two algorithms, asymptotically, have the same time complexities, **strictly speaking, the Iterative Bottom-Up has a slightly better running time (in terms of coefficients) than the Recursive with Memoization** because the former avoids function calls, and we know that function calls are costly.

We can also verify and manually compute the time complexities of these three algorithms from our pseudocodes. **Note: The manual time complexity computations are not included in this report in order to minimize the number of pages.**



Note that the time complexity of the Naïve Recursive Top-Down algorithm is, to be exact,  $O(1.6180^n)$ . Instead of showing the full computation, we will simply cite one of several sources: <https://www.ics.uci.edu/~eppstein/161/960109.html>

Coincidentally, the number 1.6180 is also the Golden-Ratio. ([https://en.wikipedia.org/wiki/Golden\\_ratio](https://en.wikipedia.org/wiki/Golden_ratio))

We can see in the “**Naïve Recursive Top-Down / Golden Ratio**” plot that the trendline (yellow line- which is this algorithm’s  $\frac{t_i}{t_i - 1}$  ratio plot) and the golden ratio (red line) consistently overlap. This supports the notion that the precise time complexity of the Naïve Recursive Top-Down algorithm is, indeed  $O(1.6180^n)$ .

On the other hand, we can see in the “**Recursive with Memoization / Golden Ratio**” and “**Iterative Bottom Up / Golden Ratio**” plots that their trendlines (yellow line) are farther from the golden ratio (red line). Hence, this explains why the  $\frac{t_i}{t_i - 1}$  ratio of these two algorithms did not, at any point, equal 1.6810 during the  $i = 0$  to 55 execution of our algorithms in this programming assignment.

**In addition:**

- Yes, the program works.
- The program is written in Java using Auburn University’s jGrasp IDE. Therefore, simply open the *M4ProgrammingAssignment.java* file in jGrasp -> then Compile  -> and then Run .

The file F is a \*.txt file but has been formatted like a \*.csv file. This way, it can be conveniently opened (and plotted) in excel.

Notes:

1. In line # 32 of the code (*M4ProgrammingAssignment.java*), the user should change the location where the log file (F) should be saved. Currently, it is set to the programmer’s local disk path/folder. **It is very strongly recommended that the user selects the same folder where he/she saved the source code *M4ProgrammingAssignment.java* file.**
2. In line # 126, the user should make sure that **the file name in line # 32 and line # 126 should exactly match.**