Colin McAllister

Collin Smith

CPSC 3283

M4 Programming Assignment

2. The naïve recursive algorithm is wildly inefficient. When Fibonacci(6) is called, Fibonacci(2) is subsequently called **5 times.**

**Algorithm 2:** The following improved recursive algorithm stores the computations in an array using -1 as a place holder. In the case of Fibonacci(6), the improved algorithm would cut down the execution time by 500%. It begins by initializing a table to the size of n. This table is stored outside of the method so that it remains intact during all the recursive calls. If this table was inside the method, each recursive call would create a table of size n, and that is unnecessary. Once the value of a certain n is calculated, it is stored inside the table at index n. If the value at index n is -1, we need to run the calculation. If not, we can return whatever value is at index n. The pseudocode below creates a table that acts as an array, but in my actual program, I chose to implement the table as a HashMap

InitializeArray(n)

1. create a public table **fib** of size **n**

2. for int **i** from 1 to n:

3. fib[i] = -1

Fibonacci(n):

1. if n == 1 **return** 0

2. if n == 2 **return** 1

3. if fib[n] != -1

4. **return** fib[n]

5. else

6. int v = Fibonacci(n-1) + Fibonacci(n-2)

7. fib[n] = v

7. **return** v

3. **Algorithm 3:** The following algorithm is an iterative way to discover the nth Fibonacci number. There are two ways to accomplish this iteratively. The first would be to use a table and store the values up to n using a for loop. The second solution uses two variables that start as 0 and 1 (the first two Fibonacci numbers) and within a for-loop, they are added together and will systematically move through the sequence of Fibonacci numbers. Once the for-loop completes n times, it will have calculated the nth Fibonacci number. The time complexity will remain the same at , but the space complexity will be in the first solution and in the second. My pseudocode will implement the second solution, and my Java code will implement the first.

ItFibonacci(n)

1. if n is 0 or 1, return n

2. int **A** = **sum** = 0, int **B** = 1

3. for i from 1 to n:

4. sum = A + B

5. B = A

6. A = sum

7. **return** sum

The Naïve Recursive Algorithm, Improved Recursive Algorithm, and Iterative Algorithm have all been implemented using Java. *The program can be compiled using a Java IDE such as IntelliJ or jGrasp and it works as intended.*

Data was collected for ranging from 0 to 55 for each of the algorithms. The time was logged and then was plotted on a graph to show if the running time growth adhered to the golden ratio. Using the classic example of a line, the golden ratio is achieved when a line is broken into two segments and , and the ratio between the larger segment to the entire line is the same as the ratio between the smaller segment and the larger segment . Or . The numbers in the Fibonacci sequence adhere to the golden ratio, since each number is the sum of the previous two numbers. The golden ratio is represented in mathematics by the equation which is roughly and is represented by . Each number in the Fibonacci sequence, when multiplied by this imaginary number will produce roughly the next number in the sequence.

Interestingly, the data from the Naïve Recursive Algorithm as seen in figure 1, when was calculated for each between 0-55, the result was consistently (see figure 4). This means that the time each loop took grew at the same rate as the numbers in the Fibonacci sequence, since each number when divided by it’s predecessor equals The improved recursive and the iterative algorithms did not produce this same result, but regardless, when was calculated, the results were constant.

The Naïve Recursive Algorithm grew at which caused it to take almost an hour to compute Fibonacci(60), but since the other two algorithms used memoization, it brought their growth complexity down to . This means that Fibonacci(>60) appeared to take only a small amount of time more than Fibonacci(55).

Figure

Figure

Figure

**Raw Data from Naïve Recursive Algorithm:**

|  |  |  |
| --- | --- | --- |
| i | T |  |
| 13 | 14600 | 1.569892 |
| 14 | 24100 | 1.650685 |
| 15 | 39000 | 1.618257 |
| 16 | 61600 | 1.579487 |
| 17 | 35200 | 0.571429 |
| 18 | 17100 | 0.485795 |
| 19 | 27900 | 1.631579 |
| 20 | 44000 | 1.577061 |
| 21 | 71200 | 1.618182 |
| 22 | 121900 | 1.712079 |
| 23 | 184900 | 1.516817 |
| 24 | 308600 | 1.66901 |
| 25 | 280000 | 0.907323 |
| 26 | 401400 | 1.433571 |
| 27 | 645300 | 1.607623 |
| 28 | 1044700 | 1.618937 |
| 29 | 1694400 | 1.621901 |
| 30 | 2740600 | 1.617446 |
| 31 | 4484400 | 1.636284 |
| 32 | 7164200 | 1.597583 |
| 33 | 11585000 | 1.617068 |
| 34 | 18753800 | 1.6188 |
| 35 | 30349000 | 1.618285 |
| 36 | 49111500 | 1.618225 |
| 37 | 79483000 | 1.618419 |
| 38 | 1.28E+08 | 1.616334 |
| 39 | 2.09E+08 | 1.627831 |
| 40 | 3.37E+08 | 1.613317 |
| 41 | 5.44E+08 | 1.613775 |
| 42 | 8.8E+08 | 1.616018 |
| 43 | 1.42E+09 | 1.61857 |
| 44 | 2.3E+09 | 1.61489 |
| 45 | 3.73E+09 | 1.619897 |
| 46 | 6.04E+09 | 1.621334 |
| 47 | 9.75E+09 | 1.613973 |
| 48 | 1.58E+10 | 1.617681 |
| 49 | 2.55E+10 | 1.618414 |
| 50 | 4.13E+10 | 1.617527 |
| 51 | 6.69E+10 | 1.620182 |
| 52 | 1.08E+11 | 1.618638 |
| 53 | 1.75E+11 | 1.618136 |

*figure 4*