Lab 1: Fibonacci

Alex Yazdani

Eric Reed

CS F003C

19 April 2023

Introduction

In this lab, three different algorithms were defined as functions in a python script. All three algorithms return the nth term of the Fibonacci series. The first algorithm directly calculates this number by creating a list, starting with terms 0 and 1, then calculating the next term by adding the previous two. The second algorithm implements recursion to find the nth term, so a list is not necessary. Instead, the function will check if n is less than or equal to 1 (that is, if the value is 0 or 1), and then proceed to call itself again, using n-1 and n-2 as inputs. The final algorithm also uses the same implementation of recursion, but has a decorator “@functools.lru\_cache()” which uses memoization to improve efficiency by removing unnecessary, repeated calculations. All three algorithms assume the user provides a valid input, so input validation is not built into the functions. The input is assumed to be a positive integer, or 0.

The “main” function in the python script acts as the test program for the three functions. Each function is tested in order to determine its complexity. For every given value of n, each test was run 1,000,000 times. This sample size was chosen for ease of understanding, as the total runtime in seconds is equivalent to the average runtime in microseconds.

This experiment was run overnight with all other applications on the system closed, using an Intel i7 core.

Direct Fibonacci Algorithm:

Data

|  |  |  |
| --- | --- | --- |
| **Size of n** | **Time to complete (uS)** | **Factor increase** |
| 50 | 14.8666109 |  |
| 100 | 30.519121 | 2.052863373 |
| 150 | 45.4639765 | 1.489688268 |
| 200 | 60.5274102 | 1.331326797 |
| 250 | 74.481159 | 1.230536029 |
| 300 | 90.8029737 | 1.219140181 |
| 350 | 106.5906997 | 1.17386794 |
| 400 | 122.6155962 | 1.150340476 |
| 450 | 148.9105251 | 1.214450117 |
| 500 | 178.8016706 | 1.200732255 |
| 550 | 197.6674453 | 1.105512295 |
| 600 | 218.5783011 | 1.105788061 |
| 650 | 235.6898796 | 1.078285806 |
| 700 | 251.9030534 | 1.068790284 |
| 750 | 272.6861985 | 1.082504538 |

Table 1: Direct Fibonacci algorithm timing data

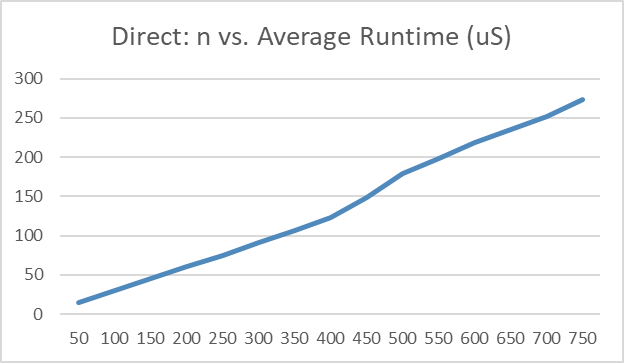
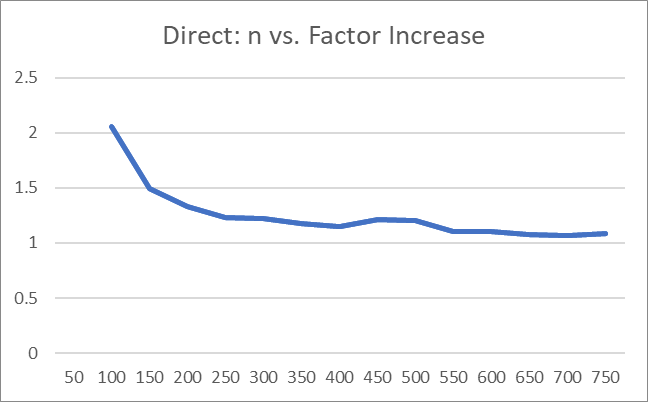
 

Figure 1.1, 1.2: Direct Fibonacci algorithm, n versus average runtime in microseconds and factor increase

Discussion

The data shows that the direct method for creating the Fibonacci function has linear time complexity. So, the empirical time complexity is O(n). Mathematically, this makes sense. For every incremental increase in n, the for loop will need to run one additional time, so there is a clear linear relationship. Ignoring any constant time requirements for each test, the mathematical time complexity is O(n).

Recursive Fibonacci Algorithm:

Data

|  |  |  |
| --- | --- | --- |
| **Size of n** | **Time to complete (uS)** | **Factor increase** |
| 1 | 0.5938348 |  |
| 2 | 1.5584281 | 2.624346199 |
| 3 | 2.5950242 | 1.665154908 |
| 4 | 4.7080964 | 1.814278418 |
| 5 | 7.9280652 | 1.683921595 |
| 6 | 12.7813273 | 1.612162233 |
| 7 | 21.1328859 | 1.653418726 |
| 8 | 34.3665045 | 1.626209722 |
| 9 | 55.9554835 | 1.62819828 |
| 10 | 91.1594074 | 1.629141626 |
| 11 | 148.895371 | 1.633351678 |
| 12 | 240.2938632 | 1.613843745 |
| 13 | 384.9959721 | 1.602188117 |
| 14 | 625.4951257 | 1.624679662 |
| 15 | 1014.442214 | 1.621822732 |
| 16 | 1635.857286 | 1.612568231 |
| 17 | 2642.543918 | 1.615387809 |
| 18 | 4276.558573 | 1.618349101 |
| 19 | 6929.83498 | 1.620423259 |
| 20 | 11248.75026 | 1.623234939 |

Table 2: Recursive Fibonacci algorithm timing data

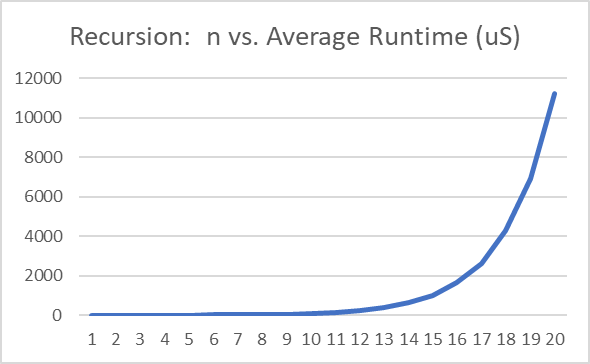
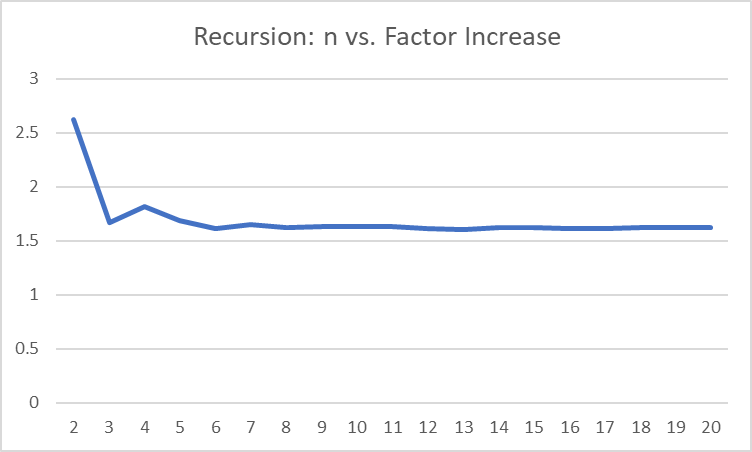
 

Figure 2.1, 2.2: Recursive Fibonacci algorithm, n versus average runtime in microseconds and factor increase

Discussion

The data for the function that uses recursion to find the nth term of the Fibonacci sequence (and does not use memoization) is visibly worse than the direct method in terms of time complexity. In fact, empirically it is shown to have exponential time complexity. The factor increase is consistently the golden ratio, ɸ = ~1.618, meaning that T(n+1) = 1.618\*T(n). So, the complexity is O(ɸ^n). Mathematically, this checks out. If we assume the initial if/else statement of the recursive function is expressed through T(n) (the base unit of time here), then the following is true:

n=0 → T(0) = T(n)

n=1 → T(1) = T(n)

n=2 → T(2) = T(1) + T(0) = 2T(n)

n=3 → T(3) = T(2) + T(1) = 2T(n) + T(n) = 3T(n)

n=4 → T(3) + T(2) = 3T(n) + 2T(n) = 5T(n)

n=5 → T(4) + T(3) = 8T(N)

n=6 → T(5) + T(4) = 13T(n)

n=7 → T(6) + T(5) = 21T(n)

n=8 → T(7) + T(6) = 34T(n)

This demonstrates the Fibonacci series, and looking at the last two terms specifically, 34/21 = ~1.619, approximately equal to the golden ratio. The mathematical time complexity is O(ɸ^n).

Recursive Fibonacci Algorithm with Memoization:

Data

|  |  |  |
| --- | --- | --- |
| **Size of n** | **Time to complete (uS)** | **Factor increase** |
| 50 | 0.1417274 |  |
| 100 | 0.1471069 | 1.037956669 |
| 150 | 0.1447348 | 0.983874992 |
| 200 | 0.1419193 | 0.98054718 |
| 250 | 0.1457278 | 1.026835674 |
| 300 | 0.1592971 | 1.093114011 |
| 350 | 0.1371419 | 0.860919 |
| 400 | 0.1467916 | 1.070362887 |
| 450 | 0.1654171 | 1.126883963 |
| 500 | 0.1397192 | 0.844647863 |
| 550 | 0.1461524 | 1.046043779 |
| 600 | 0.1259223 | 0.861582157 |
| 650 | 0.1484868 | 1.179193836 |
| 700 | 0.1333203 | 0.897859608 |
| 750 | 0.1231794 | 0.923935815 |

Table 3: Recursive/Memoized Fibonacci algorithm timing data

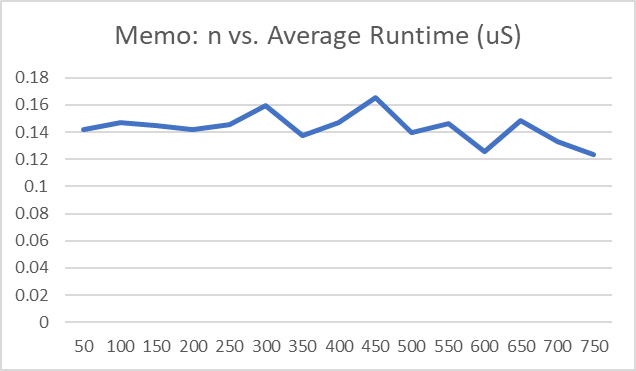
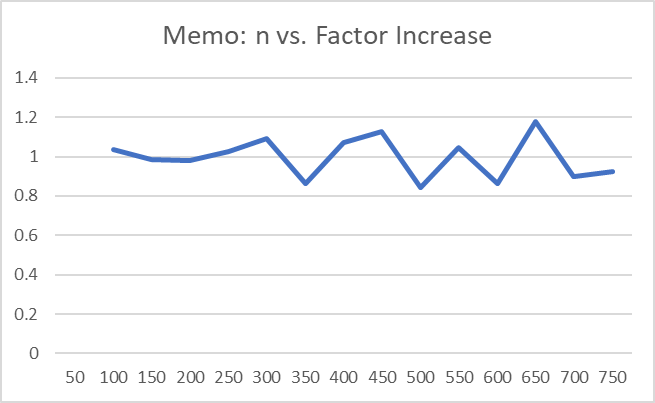
 

Figure 3.1, 3.2: Memoized Fibonacci algorithm, n versus average runtime in microseconds and factor increase

Discussion

The data for the memoized and recursive Fibonacci function shows that this algorithm has constant empirical complexity: O(1). The runtime of this function varies a small amount, but is always right around 0.14 uS. The factor increase is always between 0.8 and 1.2. Admittedly, the author of this paper is not familiar with how the “@functools.lru\_cache()” decorator operates on a detailed level. However, it is clear that by caching results, the time complexity becomes O(1). The decorator effectively turns the recursive function into a much more time-efficient iterative function, since there is no need to repeat calculations.