Duc Ngo - CS3130-001

Fibonacci Numbers: Recursive vs. Iterative

Introduction

In mathematics and computer science, an algorithm is a set of instructions that are used to accomplish a task (typically used to do something, compute a value or both). Algorithms are used for calculation, data processing, and automated reasoning. For example, we can use algorithms to find the largest number in a list, removing all the red diamond cards from a deck of playing cards, sorting a collection of names, remove duplicate elements in a database, and so on.

Algorithms are like a set of step-by-step instructions or even a recipe, containing things you need, steps to do, the order to do them, conditions to look for, and expected results. In the real world, you have performed algorithms without knowing it by that name, such as performing long division by hand which is a great example of looping over repeated steps until the problem is solved. We can also use algorithms to find the next sequence of Fibonacci Numbers.

Definition

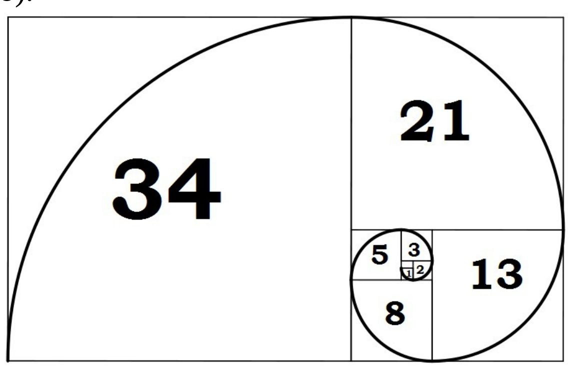
In mathematics, the Fibonacci sequence is one of the most famous formulas. Each number in the sequence is the sum of the two numbers that precede it. So, the sequence goes:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

In mathematical terms, the sequence of Fibonacci numbers is defined by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2} \qquad \text{with seed values } F_0 = 0 \text{ and } F_1 = 1$$

Fibonacci Sequence is important because it is the closest approximation in integers to the logarithmic spiral series, which follows the same rule as the Fibonacci sequence, but also the ratio of successive terms is the same (golden ratio).



Objective

The purpose of the analysis is to calculating the Fibonacci numbers recursively and iteratively and then show the theoretical order of growth of the running time for both algorithms.

Analysis

Basic Computer Information

(that was use to run recursive and iterative algorithms)

CPU: 2.5 GHz Quad-core Intel Core i5-7300HQ
GPU: NVDIA GeFore GTX 1050 Ti
RAM: 16GB DDR4
Storage: 128GB SSD and 1TB HDD

Recursive Implementation (Pseudo code)

```
F(n)
{
    if n is 0
        return 0
    else if n is 1
        return 1
    else
        return F(n-1) + F(n-2)
}
```

Recursive Implementation (Java)

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

The time complexity of recursive algorithms is:

$T(n) = T(n-1) + T(n-2)$; $T(n) \in O(\Phi)^n$; which is exponential.

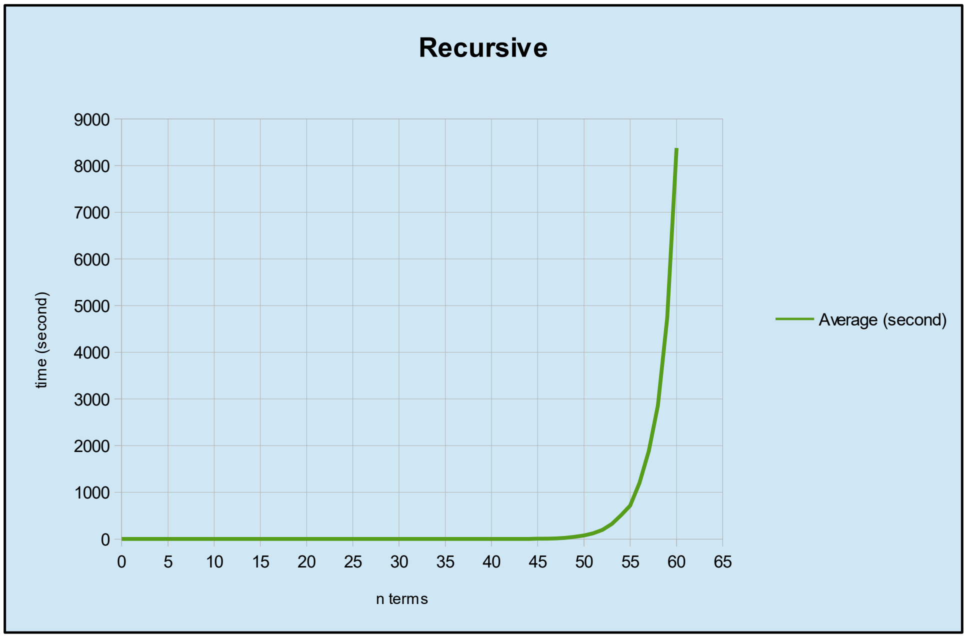
```
        F(5)
           /     \
      F(4)        F(3)
    /     \     /     \
      F(3)    F(2) F(2)   F(1)
    /     \
F(2)      F(1)
```

This implementation does a lot of repeated work. For instance, the tree above shows two computations of F(3). The second time we get to F(3), we're wasting effort computing it again, because we've already solved it once and the answer isn't going to change (this cause our algorithm to slow down as we keep recomputing the same sub-problems over and over again). Thus, this is <u>bad implementation</u> as nth get higher and we can observe this result by running three tests from 0 to 60 terms.

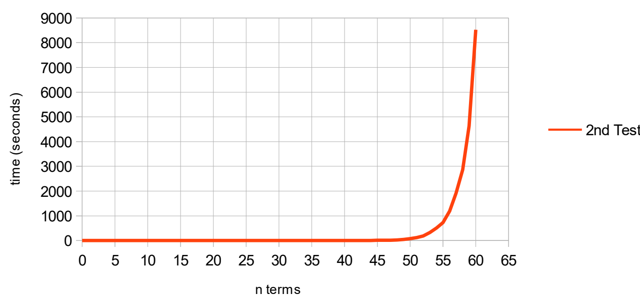
Recursive Running Test

n	1st Test (sec)	2nd Test (sec)	3rd Test (sec)	Average (sec)
0	5.33E-06	4.10E-07	8.20E-07	2.19E-06
1	8.21E-07	4.10E-07	4.10E-07	5.47E-07
2	1.23E-06	1.23E-06	1.64E-06	1.37E-06
3	1.23E-06	1.64E-06	1.23E-06	1.37E-06
4	1.64E-06	1.23E-06	1.23E-06	1.37E-06
5	1.23E-06	1.23E-06	1.23E-06	1.23E-06
6	1.23E-06	4.51E-06	1.23E-06	2.33E-06
7	4.51E-06	9.27E-05	2.46E-06	3.32E-05
8	1.15E-05	1.23E-06	8.21E-07	4.51E-06
9	1.23E-06	8.20E-07	8.21E-07	9.57E-07
10	1.23E-06	1.23E-06	8.20E-07	1.09E-06
11	1.64E-06	1.23E-06	1.64E-06	1.50E-06
12	2.05E-06	2.05E-06	2.05E-06	2.05E-06
13	2.87E-06	2.87E-06	2.22E-05	9.30E-06
14	4.92E-06	4.51E-06	7.39E-06	5.61E-06
15	0.000007795	0.000026667	0.000003693	1.27E-05
16	0.000005334	0.000006153	0.000006154	5.88E-06
17	0.000009026	0.000008616	0.000029128	1.56E-05
18	0.000013949	0.000014359	0.000014359	1.42E-05
19	0.000022154	0.000024616	0.000020102	2.23E-05
20	0.000033231	0.000033231	0.000034461	3.36E-05
21	0.000053333	0.000054564	0.000052923	5.36E-05
22	0.000085744	0.000085333	0.000085333	8.55E-05
23	1.38E-04	1.37E-04	1.38E-04	1.38E-04
24	2.24E-04	2.23E-04	2.24E-04	2.24E-04
25	4.64E-04	3.85E-04	3.92E-04	4.14E-04
26	6.17E-04	6.51E-04	6.67E-04	6.45E-04
27	1.04E-03	1.08E-03	1.10E-03	1.08E-03
28	1.56E-03	1.65E-03	1.99E-03	1.73E-03
29	5.85E-03	3.27E-03	2.79E-03	3.97E-03

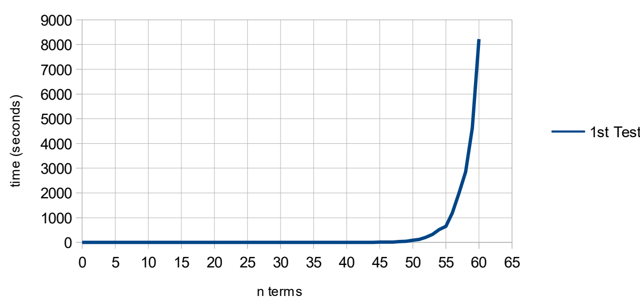
n	1^{st} Test (sec)	2^{nd} Test (sec)	3^{rd} Test (sec)	Average (sec)
30	6.00E-03	4.42E-03	4.10E-03	4.84E-03
31	8.04E-03	6.88E-03	7.90E-03	7.61E-03
32	1.09E-02	1.07E-02	1.05E-02	1.07E-02
33	1.75E-02	1.75E-02	2.00E-02	1.84E-02
34	3.06E-02	3.06E-02	3.16E-02	3.09E-02
35	5.00E-02	5.57E-02	5.40E-02	5.32E-02
36	7.89E-02	9.24E-02	8.41E-02	8.51E-02
37	0.146043777	0.132344507	0.131907584	1.37E-01
38	0.221109563	0.205667933	0.2070066	2.11E-01
39	0.361603806	0.340136747	0.352909249	3.52E-01
40	0.585958084	0.568388046	0.565568356	5.73E-01
41	0.981213144	0.883465532	0.94953399	9.38E-01
42	1.495325132	1.514393013	1.516228499	1.51E+00
43	2.400842748	2.381675995	2.357899193	2.38E+00
44	3.895917214	3.881710866	3.82461758	3.87E+00
45	6.385151274	6.576861886	6.226686583	6.40E+00
46	10.339959105	10.505587688	10.43309626	1.04E+01
47	16.49564308	15.998322462	16.349713764	1.63E+01
48	28.980937452	27.019004702	27.049175344	2.77E+01
49	46.477638172	48.398749211	44.586881159	4.65E+01
50	81.677605597	75.543269298	73.620449953	7.69E+01
51	116.540178839	120.339447927	124.808465798	1.21E+02
52	204.855063503	188.59708628	201.219489872	1.98E+02
53	324.642708396	325.866597443	323.783358126	3.25E+02
54	521.293395862	511.900452595	506.201791014	5.13E+02
55	652.980852529	722.598125458	775.982928835	7.17E+02
56	1186.680916364	1188.314382511	1227.99022688	1.20E+03
57	1989.698845785	1909.979937247	1769.683498516	1.89E+03
58	2851.514045527	2861.246010465	2861.028696592	2.86E+03
59	4625.038523045	4633.668922939	4951.388375886	4.74E+03
60	8226.26125517	8528.293395862	8372.808465798	8.38E+03



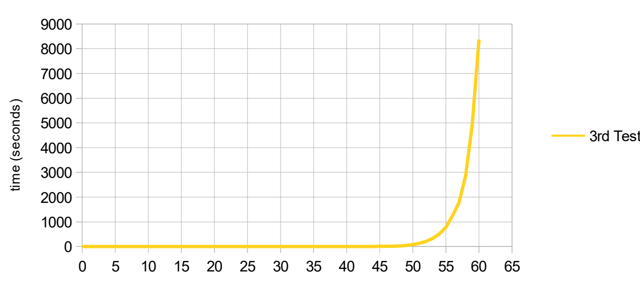
Recursive - 2nd Test



Recursive - 1st Test



Recursive - 3rd Test



*Base on the experimental results, we can conclude that recursive function is $O(\Phi)^n$.

We can avoid the repeated work done in the recursive method by storing the previous two numbers only (because that is all we need to get the next Fibonacci number in series). This will significantly reduce the running time.

Iterative Implementation (Pseudo code)

```
F(n)
{
    if n is 0
        return 0
    if n is 1
        return 1

    prevPrev = 0
    prev = 1
    result = 0

    for i = 2 to n
        result = prev + prevPrev
        prevPrev = prev
        prev = result
    return result
}
```

Iterative Implementation (Java)

```java
public static long F(int n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;

    long prevPrev = 0;
    long prev = 1;
    long result = 0;

    for (int i = 2; i <= n; i++)
    {
        result = prev + prevPrev;
        prevPrev = prev;
        prev = result;
    }
    return result;
}
```

The time complexity of iterative algorithms is: $T(n) \in O(n)$

$$C_1 + C_2 + C_3 + C_4 + C_5 + C_6 + C_7 + C_8(n) + C_9(n) + C_{10}(n) + C_{11}(n) + C_{12} =$$

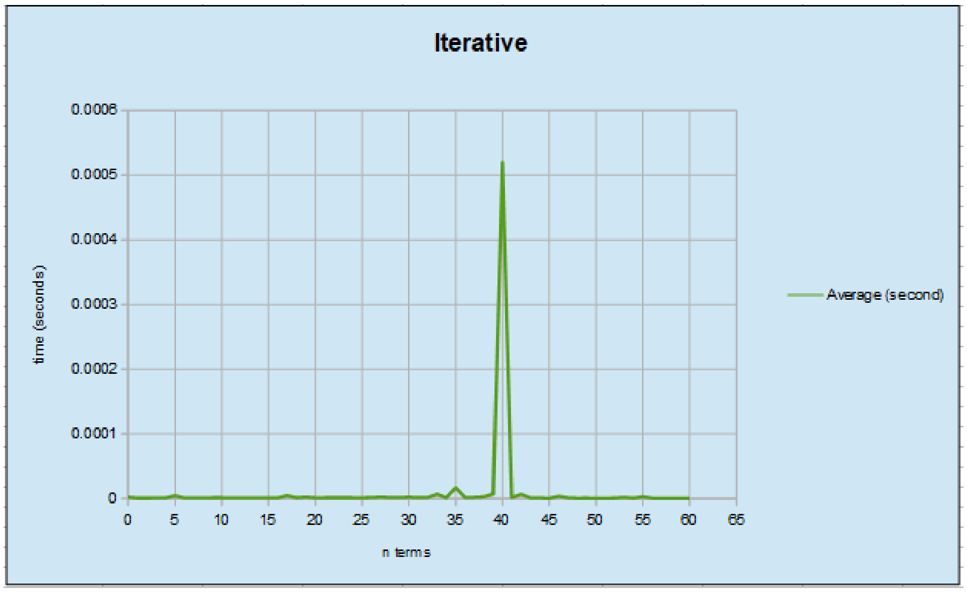
$$= n(C_8 + C_9 + C_{10} + C_{11}) + (C_1 + C_2 + C_3 + C_4 + C_5 + C_6 + C_7 + C_{12}) \in O(n)$$

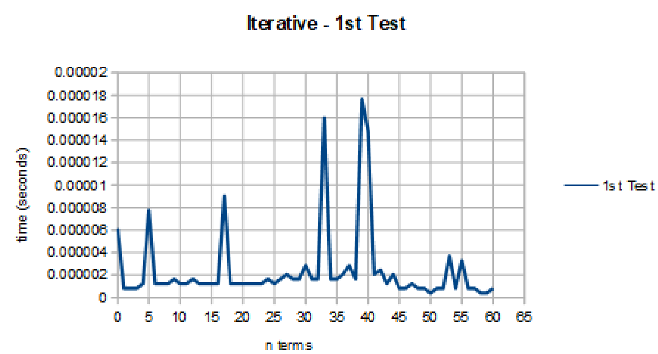
		cost	# of times
1	**if** n **is** 0	C_1	1
2	return 0	C_2	1
3	**if** n **is** 1	C_3	1
4	return 1	C_4	1
5	prevPrev = 0	C_5	1
6	prev = 1	C_6	1
7	result = 0	C_7	1
8	**for** i = 2 **to** n	C_8	(n - 1) + 1 = n
9	result = prev + prevPrev	C_9	n
10	prevPrev = prev	C_{10}	n
11	prev = result	C_{11}	n
12	**return** result	C_{12}	1

Iterative Running Test

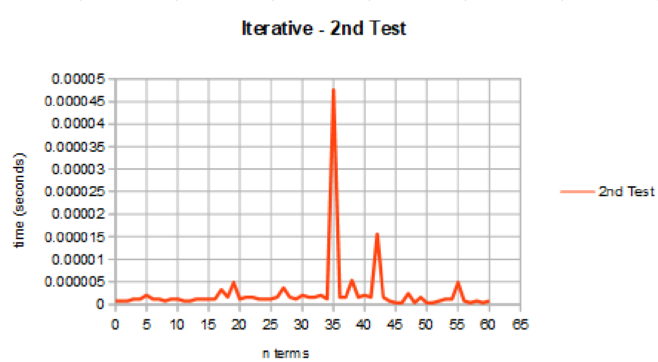
n	1st Test (sec)	2nd Test (sec)	3rd Test (sec)	Average (sec)
0	0.000006154	0.000000821	0.000000411	0.000002462
1	0.000000821	0.000000821	0.00000082	8.2066667E-007
2	0.00000082	0.00000082	0.000000821	8.2033333E-007
3	0.00000082	0.000001231	0.00000123	1.0936667E-006
4	0.000001231	0.000001231	0.000001231	0.000001231
5	0.000007794	0.000002051	0.000003282	4.3756667E-006
6	0.000001231	0.000001231	0.00000123	1.2306667E-006
7	0.000001231	0.00000123	0.00000123	1.2303333E-006
8	0.000001231	0.000000821	0.00000082	9.5733333E-007
9	0.000001641	0.000001231	0.000001231	1.3676667E-006
10	0.00000123	0.000001231	0.000001641	1.3673333E-006
11	0.00000123	0.000000821	0.00000082	0.000000957
12	0.000001641	0.000000821	0.000001231	0.000001231
13	0.000001231	0.000001231	0.000001231	0.000001231
14	0.00000123	0.000001231	0.000001641	1.3673333E-006
15	0.000001231	0.000001231	0.000001231	0.000001231
16	0.000001231	0.000001231	0.000001641	1.3676667E-006
17	0.000009026	0.000003282	0.000002051	4.7863333E-006
18	0.000001231	0.000001641	0.000001231	1.3676667E-006
19	0.00000123	0.000004923	0.000001231	2.4613333E-006
20	0.000001231	0.000001231	0.000001231	0.000001231
21	0.00000123	0.000001641	0.000001231	1.3673333E-006
22	0.00000123	0.000001641	0.000001641	0.000001504
23	0.000001231	0.000001231	0.000001641	1.3676667E-006
24	0.000001641	0.000001231	0.000001231	1.3676667E-006
25	0.000001231	0.000001231	0.00000123	1.2306667E-006
26	0.000001641	0.000001641	0.000001641	0.000001641
27	0.000002051	0.000003692	0.000001641	2.4613333E-006
28	0.000001641	0.000001641	0.000001641	0.000001641
29	0.000001641	0.000001231	0.000001641	1.5043333E-006

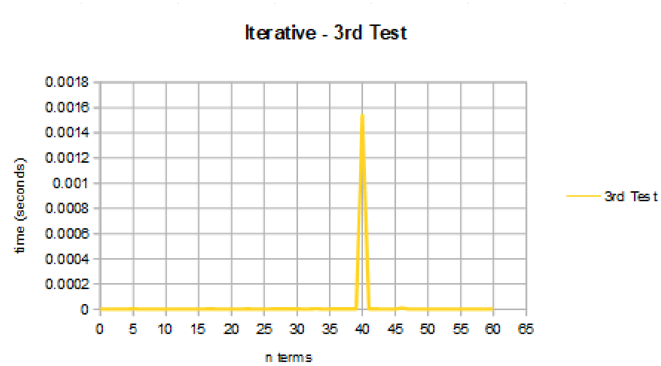
n	1st Test (sec)	2nd Test (sec)	3rd Test (sec)	Average (sec)
30	0.000002872	0.000002051	0.000002052	0.000002325
31	0.000001641	0.000001641	0.000001231	1.50433333E-006
32	0.000001641	0.000001641	0.000001641	0.000001641
33	0.000016	0.000002051	0.000002872	6.97433333E-006
34	0.000001641	0.000001231	0.000001231	1.36766667E-006
35	0.000001641	0.00004759	0.000001641	1.69573333E-005
36	0.000002051	0.000001641	0.000001641	1.77766667E-006
37	0.000002871	0.000001641	0.000001641	0.000002051
38	0.000001641	0.000005334	0.000001641	0.000002872
39	0.000017641	0.000001641	0.000002052	7.11133333E-006
40	0.000014769	0.000002051	0.001540512	0.0005191107
41	0.000002051	0.000001641	0.000001641	1.77766667E-006
42	0.000002462	0.000015589	0.000002051	6.70066667E-006
43	0.00000123	0.000001641	0.00000082	1.23033333E-006
44	0.000002051	0.00000082	0.00000041	1.09366667E-006
45	0.000000821	0.000000411	0.00000041	5.47333333E-007
46	0.000000821	0.00000041	0.000009847	3.69266667E-006
47	0.00000123	0.000002461	0.00000082	1.50366667E-006
48	0.00000082	0.00000041	0.000000821	6.83666667E-007
49	0.00000082	0.000001641	0.00000041	0.000000957
50	0.00000041	0.00000041	0.00000041	0.00000041
51	0.00000082	0.000000411	0.00000041	0.000000547
52	0.00000082	0.00000082	0.00000123	9.56666667E-007
53	0.000003692	0.000001231	0.000001231	2.05133333E-006
54	0.000000821	0.000001231	0.00000041	8.20666667E-007
55	0.000003282	0.000004923	0.000000411	0.000002872
56	0.000000821	0.000000821	0.00000082	8.20666667E-007
57	0.000000821	0.00000041	0.00000041	0.000000547
58	0.00000041	0.00000082	0.00000041	5.46666667E-007
59	0.00000041	0.00000041	0.00000041	0.00000041
60	0.000000821	0.000000821	0.000000411	6.84333333E-007





*Even though the graph doesn't look linear due to n = 40, our range never reaches past the 1-second mark thus we can safely assume the iterative function is linear.





*Base on the experimental results, we can conclude that iterative function is O(n).

Conclusion

Ranking: $O(n)$, $O(\Phi)^n$

In conclusion, the iterative algorithm grows much slower since it is a linear function, which results in faster running time. While recursive algorithm grows very quickly since it is an exponential function, which result in longer running time.

In this application, it is not recommended to use recursion to find our next Fibonacci Numbers. Recursion uses a lot of memory if we make too many calls (i.e. calling 60 times). While doing it in iteratively, such as looping, it is more memory efficient. The downside of loops is that loops instances are cleared after every iteration which is not great for backtracking.