Finding the N^{th} Fibonacci Number

COT6405 Project by Nick Petty

Table of Contents

Abstract

The Fibonacci Numbers are a sequence of positive integers defined by a recursive relationship. That is, $F_n = F_{n-1} + F_{n-2}$, where $F_0 = 0$ and $F_1 = 1$. Likely owning to its simplicity, the Fibonacci Numbers appear frequently in the natural world and a wide range sciences. However, the value of F_n grows very quickly, and the recursive method described becomes impossibly slow as n increases. This paper will discuss some uses of Fibonacci Numbers, then examine two methods of finding the n^{th} Fibonacci Number, given n. The two methods examined will be the previously described recursive approach and the dynamic programming approach. By comparing the theoretical and experimental runtimes of both approaches, dynamic programming will be shown to be the better method.

Problem Description

Given an integer, $n > 1$, the value of the n^{th} Fibonacci Number, F_n, is sought. This is found by applying the recurrence relation $F_n = F_{n-1} + F_{n-2}$, where $F_0 = 0$ and $F_1 = 1$. There is discrepancy in this definition, as some problem descriptions may choose $F_1 = 0$ and $F_2 = 1$, or $F_1 =$ and $F_2 = 1$, or other values and labels for the first two terms. As the values of n increase, the values of F_n grow rapidly and calculation becomes very time and space intensive. Because the Fibonacci Numbers have a wide array of uses and appear frequently in many scientific fields, an efficient method of calculating larger members of this set is necessary.

Practical Applications

Leonardo of Pisa, commonly known as Fibonacci, introduced a series of recursive numbers to Western mathematics in his book Liber Abaci, published in 1202. It wasn't until the 19^{th} century, though, that number theorist Édouard Lucas gave the Fibonacci numbers their name (Knott). The concept of Fibonacci Numbers, and their generating function, had been known to mathematics in other parts of the world as well. Donald Knuth cites Indian scholars and their writings, saying:

> "Before Fibonacci wrote his work, the sequence F_n had already been discussed by Indian scholars, who had long been interested in rhythmic patterns... both Gopala (before 1135 AD) and Hemachandra (c. 1150) mentioned the numbers 1,2,3,5,8,13,21 explicitly." (p. 100)

Appearing frequently in mathematics and science for centuries, it is clear that these numbers are important and have intriguing uses. This section will discuss a few common applications of the Fibonacci Numbers and the general rule that it follows.

The Golden Ratio, φ, is a number that often results when establishing a "beautiful" or "natural" ratio between two measurements. Mathematically, two numbers exhibit this relationship if their ratio is the same as the ratio of their sum to the larger of the two (Wikipedia). That is,

$$\frac{a}{b} = \frac{a+b}{a} = \phi \approx 1.618$$

This ratio was used frequently in classical architecture and it is observed in many instances of biological growth patterns. The value of φ can be calculated by taking any sufficiently large Fibonacci numbers, $F_n = F_{n-1} + F_{n-2}$, where $F_{n-1} = a$ and $F_{n-2} = b$, as shown:

$$\frac{a}{b} = \frac{F_{n-1}}{F_{n-2}} = \frac{a+b}{a} = \frac{F_{n-1} + F_{n-2}}{F_{n-1}} = \frac{F_n}{F_{n-1}} = \phi$$

Taking n = 25, for example, the calculation is:

$$\frac{F_{n-1}}{F_{n-2}} = \frac{46368}{28657} \approx 1.618 = \phi$$

$$\frac{F_n}{F_{n-1}} = \frac{75025}{46368} \approx 1.618 = \phi$$

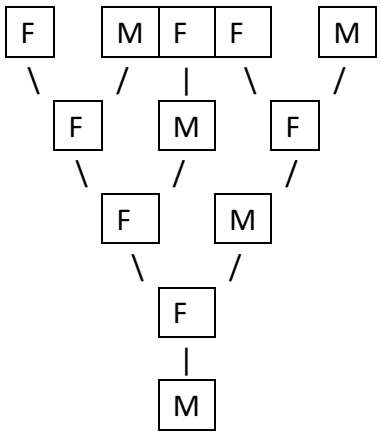
If a highly precise value of φ is needed, large Fibonacci Numbers can be used to calculate it.

In computer science, the heap data structure provides an ordering of its elements such that maximum or minimum values can be quickly accessed. Chapter 19 of the CLRS book is dedicated to a special heap implementation, called the Fibonacci Heap, whose value is described in the opening paragraph:

> "The Fibonacci heap data structure serves a dual purpose. First, it supports a set of operations that constitutes what is known as a "mergeable heap." Second, several Fibonacci-heap operations run in constant amortized time, which makes this data structure well suited for applications that invoke these operations frequently." (p. 505)

The Fibonacci heap is actually a collection of trees that are min-heap ordered, and whose roots are connected in a list. It was named after Fibonacci because the numbers in the series appear in the runtime analysis of the data structure. Specifically, the Fibonacci heap uses a forest of trees, given an order, n, such that the number of nodes in the trees is $\geq F_{n+2}$ (Growing the Web). Operational runtime depends on summing the numbers of nodes in the trees, which is a set of Fibonacci numbers.

In nature, the Fibonacci numbers are found in what is popularly called the "Bee Ancestry Code." A point of bee biology is that female bees produce males when unfertilized and females when fertilized (Bee Ancestry). From this, tracing 1 male bee's ancestry shows he has 1 parent, 2 grandparents, 3 great-grandparents, 5 great-great-grandparents, and so on, which are the Fibonacci numbers. The following table illustrates that n generations back, a male bee has F_n ancestors:



Generation 5: 3 + 2 = 5 bees F_5

Generation 2: 2 + 1 = 3 bees F_4

Generation 3: 1 + 1 = 2 bees F_3

Generation 2: 1 + 0 = 1 bee F_2

Generation 1: 1 bee F_1

The Fibonacci Numbers are used frequently in the sciences, so much so that a dedicated research community has developed around them. This is the Fibonacci Association, and it publishes a journal called the Fibonacci Quarterly, which is at www.fq.math.ca. The journal has been in regular publication since 1963, with new volumes released four times per year. The Fibonacci Association also regularly hosts conferences around the world focused on recent research related to the Fibonacci Numbers and sequence. Considering such long-running work dedicated to applications of Fibonacci Numbers, it's clear that they have great importance in mathematics and beyond.

Competing Algorithms

Two methods of calculating the nth Fibonacci Number are implemented, one done recursively and one done with dynamic programming. By running the two methods against the same values of n, their comparative execution speeds are measured. Even with small values of n, it is quickly apparent that the recursive method is unsuitable for regular use if Fibonacci numbers are needed.

The recursive algorithm is the simpler of the two, requiring only a base case and a recursive step. This is the pseudo code, from GeeksforGeeks:

```
recursive_fib (int n)
      if n ≤ 1
            return n
      return recursive_fib(n-1) + recursive_fib(n-2)
```

The single comparison and three arithmetic operations all take constant time, c, so the recurrence relation $T(n) = T(n-1) + T(n-2) + c$ is used to determine the theoretical runtime (Mycodeschool). To find an upper bound, observe that $T(n-2) \leq T(n-1)$, but they are close enough that $T(n-2) \approx T(n-1)$ can be estimated. In this case, the recurrence relation becomes $T(n) = 2T(n-1) + c$, and backwards substitution can be applied:

$T(n) = 2T(n-1) + c$

$= 2(2T(n-2) + c) + c = 4T(n-2) + 3c$

$= 2(4T(n-3) + 3c) + c = 8T(n-3) + 7c$

$= 2(8T(n-4) + 7c) + c = 16T(n-4) + 15c$

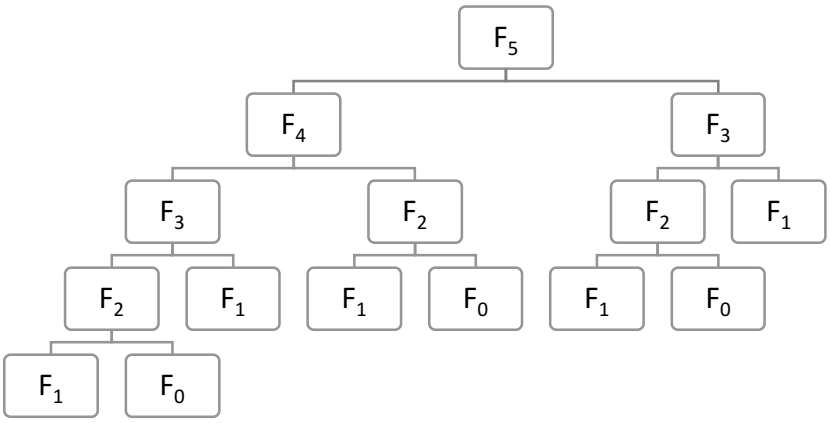
$= 2^kT(n-k) + (2^k-1)c$

Since $T(0) = 0 \rightarrow n - k = 0 \rightarrow k = n$, and then

$T(n) = 2^nT(0) + (2^n - 1)c = 2^nc - c$

$T(n) = O(2^n)$

This exponential runtime is the result of repeatedly calculating the same numbers because the recursion does not stop until it reaches F_0 or F_1. The recursion tree created by `recursive_fib(5)` is shown below (GeeksforGeeks). In this example, there are 15 recursive calls, which find F_0 three times, F_1 five times, F_2 three times, F_3 two times, and F_4 once before returning F_5. Because of this large number of repeated calculations, the recursive implementation of the n^{th} Fibonacci Number runs impossibly slow.



By using dynamic programming to calculate the n^{th} Fibonacci Number, the excessive work done by recursion is avoided. This implementation creates an array and stores the initial sequence values of F_0 and F_1, then iterates up to n while filling the array with calculated Fibonacci Numbers. The index of each element in the array corresponds to its position in the sequence, with F_n being the last element added at the largest index. The pseudo code is somewhat longer than the recursive version, but still very short and straightforward (GeeksforGeeks):

```
dynamic_fib (int n)
      int fib_array[n+1]
      fib_array[0] = 0
      fib_array[1] = 1
      for i = 2 to n
            fib_array[i] = fib_array[i-1] + fib_array[i-2]
      return fib_array[n]
```

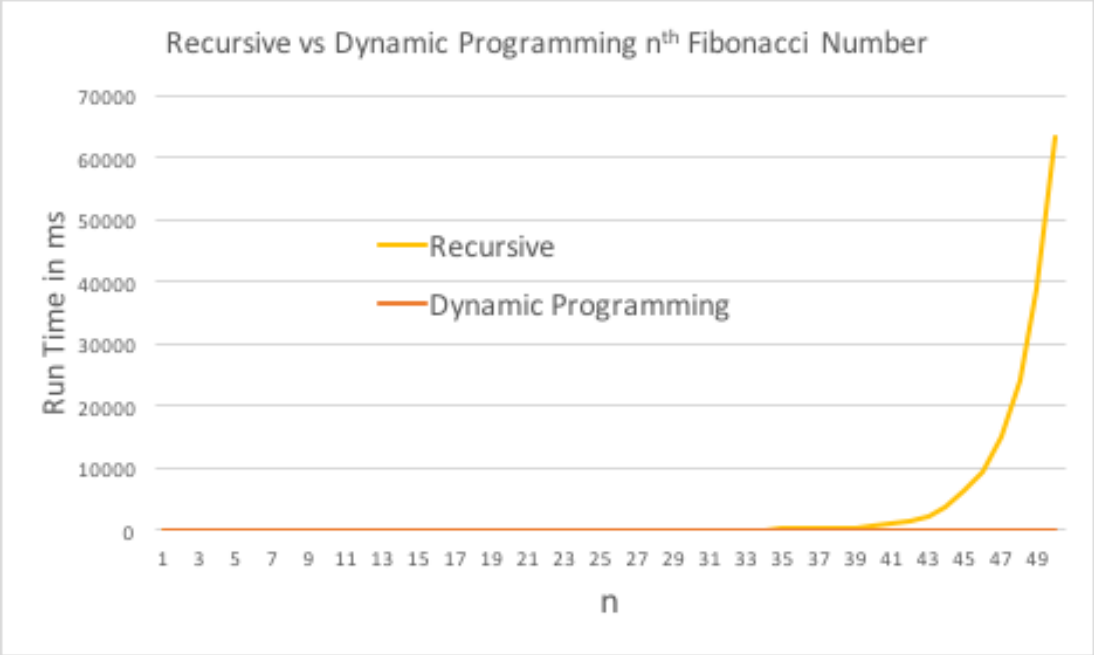
This method is initialized with three constant time operations, and has only a single `for` loop that traverses the n elements of the array. At each element, an operation is performed, but this runs in constant time and does not increase overall run time. The total theoretical runtime for this method is therefore O(n).

With one method of calculating the nth Fibonacci Number running at O(2n) time compared to another running at O(n) time, it is clear that the recursive implementation is vastly inferior to the dynamic programming version. The following experimental section will prove this with actual program execution times.

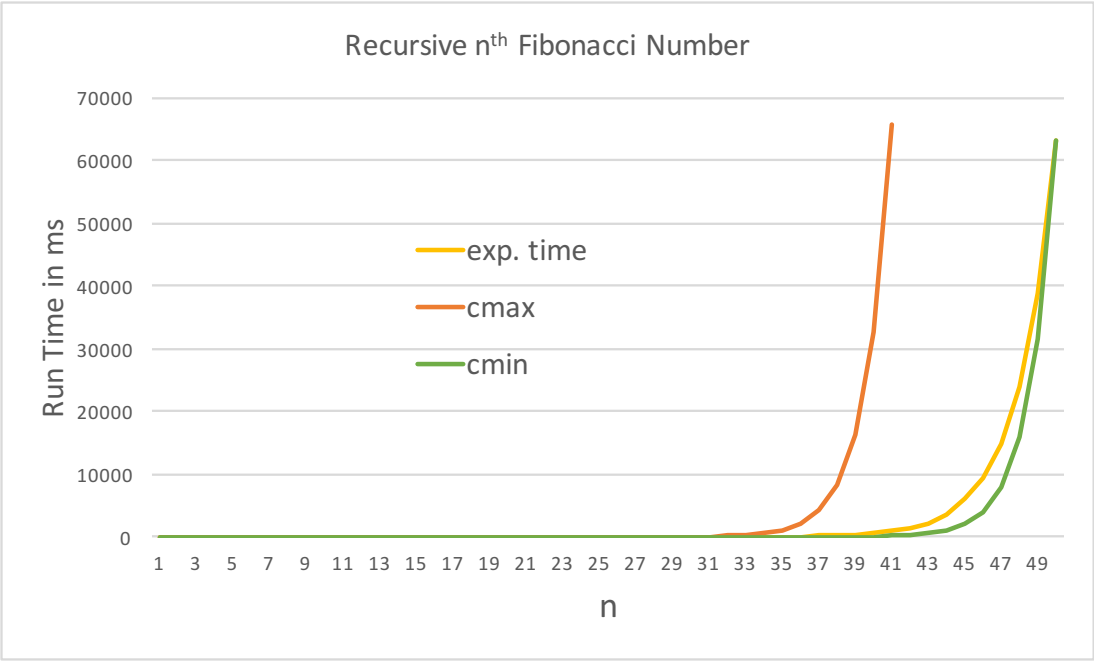
Experiments

The first step in comparing the efficiency of the recursive algorithm to the dynamic programming algorithm is to make a direct measurement of their performance on the same data set. In the graph below, the first 50 Fibonacci Numbers are calculated with both versions of the algorithm and the runtimes for each value of n are recorded in milliseconds. This is 100 total runs. Clearly, the dynamic programming approach is superior, as

the recursive implementation quickly tends towards impossibly long runtimes. However, the previously stated theoretical runtimes must be proven accurate as well, so further investigation is required.



Examining the recursive implementation on its own, verification of the theoretical runtime of $O(2^n)$ must be demonstrated. To show this, two constants, c_{min} and c_{max} are found such that $c_{min}2^n \leq$ experimental runtime $\leq c_{max}2^n$. These values of c are calculated by taking the ratio of experimental runtime to theoretical runtime for each run, then finding the maximum and minimum of this set. To estimate theoretical runtime, the time to return F_0 and F_1 is assumed to be 0.001ms and 0.002ms, respectively. The following table shows the value of n for each run, the computed Fibonacci Number, the actual runtime, the theoretical runtime, and the constant c. The values of c_{min} and c_{max} are determined, then $c_{min}2^n$ and $c_{max}2^n$ are calculated, which provides the bounding for the experimental runtime. Graphing these points shows that the recursive implementation of the n^{th} Fibonacci Number is indeed $O(2^n)$.

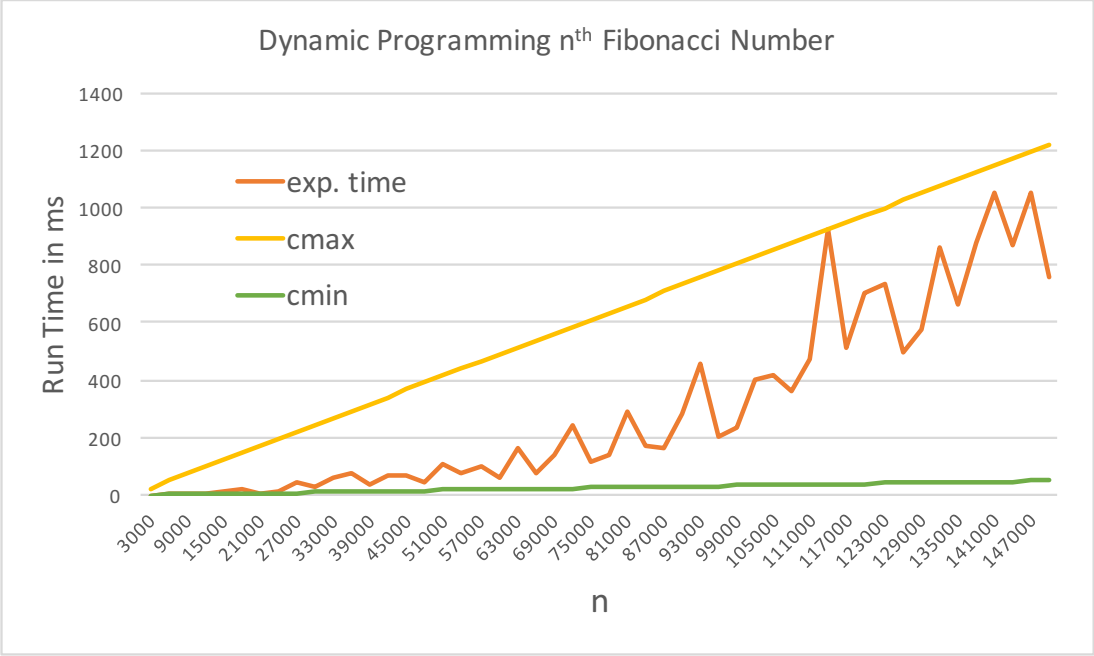


Recursive						c_{max}	c_{min}
n	value	exp. time	thr. time	c		2.98023E-05	5.60E-08
1	1	0	0.002	0		5.96046E-08	1.1203E-10
2	1	0	0.004	0		1.19209E-07	2.2406E-10
3	2	0	0.008	0		2.38419E-07	4.4811E-10
4	3	0	0.016	0		4.76837E-07	8.9622E-10
5	5	0	0.032	0		9.53674E-07	1.7924E-09
6	8	0	0.064	0		1.90735E-06	3.5849E-09
7	13	0	0.128	0		3.8147E-06	7.1698E-09
8	21	0	0.256	0		7.62939E-06	1.434E-08
9	34	0	0.512	0		1.52588E-05	2.8679E-08
10	55	0	1.024	0		3.05176E-05	5.7358E-08
11	89	0	2.048	0		6.10352E-05	1.1472E-07
12	144	0	4.096	0		0.00012207	2.2943E-07
13	233	0	8.192	0		0.000244141	4.5887E-07
14	377	0	16.384	0		0.000488281	9.1773E-07
15	610	0	32.768	0		0.000976563	1.8355E-06
16	987	0	65.536	0		0.001953125	3.6709E-06
17	1597	0	131.072	0		0.00390625	7.3418E-06
18	2584	0	262.144	0		0.0078125	1.4684E-05
19	4181	0	524.288	0		0.015625	2.9367E-05
20	6765	0	1048.576	0		0.03125	5.8735E-05
21	10946	0	2097.152	0		0.0625	0.00011747
22	17711	0	4194.304	0		0.125	0.00023494
23	28657	0	8388.608	0		0.25	0.00046988
24	46368	0	16777.216	0		0.5	0.00093976
25	75025	1	33554.432	2.98023E-05		1	0.00187951
26	121393	1	67108.864	1.49012E-05		2	0.00375903
27	196418	1	134217.728	7.45058E-06		4	0.00751805
28	317811	1	268435.456	3.72529E-06		8	0.01503611
29	514229	2	536870.912	3.72529E-06		16	0.03007221
30	832040	5	1073741.82	4.65661E-06		32	0.06014442
31	1346269	7	2147483.65	3.25963E-06		64	0.12028885
32	2178309	11	4294967.3	2.56114E-06		128	0.2405777
33	3524578	17	8589934.59	1.97906E-06		256	0.4811554
34	5702887	30	1.72E+07	1.74623E-06		512	0.96231079
35	9227465	46	3.44E+07	1.33878E-06		1024	1.92462158
36	14930352	76	6.87E+07	1.10595E-06		2048	3.84924316
37	24157817	124	1.37E+08	9.02219E-07		4096	7.69848633
38	39088169	205	2.75E+08	7.45786E-07		8192	15.3969727
39	63245986	317	5.50E+08	5.7662E-07		16384	30.7939453
40	102334155	509	1.10E+09	4.62933E-07		32768	61.5878906
41	165580141	834	2.20E+09	3.79259E-07		65536	123.175781
42	267914296	1304	4.40E+09	2.96495E-07		131072	246.351563
43	433494437	2230	8.80E+09	2.53522E-07		262144	492.703125
44	701408733	3459	1.76E+10	1.96621E-07		524288	985.40625
45	1134903170	6106	3.52E+10	1.73543E-07		1048576	1970.8125
46	1836311903	9237	7.04E+10	1.31266E-07		2097152	3941.625
47	2971215073	14771	1.41E+11	1.04954E-07		4194304	7883.25
48	4807526976	23943	2.81E+11	8.50626E-08		8388608	15766.5
49	7778742049	38622	5.63E+11	6.86065E-08		16777216	31533
50	1.2586E+10	63066	1.13E+12	5.60139E-08		33554432	63066

Moving onto the dynamic programming implementation, the theoretical runtime is O(n), and proving this experimentally follows the same methodology that was done for the recursive implementation. This time, $c_{min}n \leq$ experimental runtime $\leq c_{max}n$ is the bounding function, however. From the graph to the right, it is clear that the first 50 Fibonacci numbers do not provide meaningful data regarding experimental runtimes. Only occasionally does the runtime exceed 0ms, so the values of c are worthless.

Dynamic Programming				
n	value	exp. time	thr. time	c
1	1	1	0.001	1000
2	1	0	0.002	0
3	2	0	0.003	0
4	3	0	0.004	0
5	5	0	0.005	0
6	8	0	0.006	0
7	13	0	0.007	0
8	21	0	0.008	0
9	34	0	0.009	0
10	55	0	0.01	0
11	89	0	0.011	0
12	144	0	0.012	0
13	233	0	0.013	0
14	377	0	0.014	0
15	610	0	0.015	0
16	987	0	0.016	0
17	1597	0	0.017	0
18	2584	0	0.018	0
19	4181	1	0.019	52.6315789
20	6765	0	0.02	0
21	10946	0	0.021	0
22	17711	0	0.022	0
23	28657	0	0.023	0
24	46368	0	0.024	0
25	75025	0	0.025	0
26	121393	0	0.026	0
27	196418	0	0.027	0
28	317811	0	0.028	0
29	514229	0	0.029	0
30	832040	0	0.03	0
31	1346269	0	0.031	0
32	2178309	0	0.032	0
33	3524578	0	0.033	0
34	5702887	0	0.034	0
35	9227465	1	0.035	28.5714286
36	14930352	0	0.036	0
37	24157817	0	0.037	0
38	39088169	0	0.038	0
39	63245986	0	0.039	0
40	102334155	0	0.04	0
41	165580141	0	0.041	0
42	267914296	0	0.042	0
43	433494437	0	0.043	0
44	701408733	0	0.044	0
45	1134903170	0	0.045	0
46	1836311903	0	0.046	0
47	2971215073	0	0.047	0
48	4807526976	0	0.048	0
49	7778742049	0	0.049	0
50	1.2586E+10	0	0.05	0

To get meaningful data, a larger investigation space is needed. For this, the values of n were calculated up to 150,000 – that is, $F_{150,000}$ – at which point asymptotic behavior could be observed. The graph and table below show the results of this modified experiment. Here, the first Fibonacci Number calculated was $F_{3,000}$, and its theoretical runtime was assumed to be 3ms, which was quite accurate. From there, the next tested value of n steps 3,000 each time, giving a total of 50 runs from n = 3,000 to n = 150,000. The values of c_{min} and c_{max} found on these runs were able to show that the experimental runtimes were bounded by a constant, c, multiplied by the theoretical runtimes. Therefore, the dynamic programming implementation has a runtime of O(n).



Dynamic Programming with Big Integers						c_{max}	c_{min}
n	value	exp. time	thr. time	c		8.13157895	0.33333333
3000	4106158863	1	3	0.33333333		24.3947368	1
6000	3770131493	6	6	1		48.7894737	2
9000	3461602912	4	9	0.44444444		73.1842105	3
12000	3178322757	7	12	0.58333333		97.5789474	4
15000	2918224824	10	15	0.66666667		121.973684	5
18000	2679411996	17	18	0.94444444		146.368421	6
21000	2460142407	9	21	0.42857143		170.763158	7
24000	2258816738	15	24	0.625		195.157895	8
27000	2073966548	42	27	1.55555556		219.552632	9
30000	1904243567	27	30	0.9		243.947368	10
33000	1748409860	64	33	1.93939394		268.342105	11
36000	1605328799	75	36	2.08333333		292.736842	12
39000	1473956772	36	39	0.92307692		317.131579	13
42000	1353335570	67	42	1.5952381		341.526316	14
45000	1242585401	69	45	1.53333333		365.921053	15
48000	1140898467	48	48	1		390.315789	16
51000	1047533080	106	51	2.07843137		414.710526	17
54000	9618082471	77	54	1.42592593		439.105263	18
57000	8830987023	98	57	1.71929825		463.5	19
60000	8108303504	61	60	1.01666667		487.894737	20
63000	7444760766	166	63	2.63492063		512.289474	21
66000	6835519025	79	66	1.1969697		536.684211	22
69000	6276134561	140	69	2.02898551		561.078947	23
72000	5762527305	243	72	3.375		585.473684	24
75000	5290951081	113	75	1.50666667		609.868421	25
78000	4857966282	141	78	1.80769231		634.263158	26
81000	4460414778	287	81	3.54320988		658.657895	27
84000	4095396888	170	84	2.02380952		683.052632	28
87000	3760250225	162	87	1.86206897		707.447368	29
90000	3452530277	279	90	3.1		731.842105	30
93000	3169992581	458	93	4.92473118		756.236842	31
96000	2910576347	203	96	2.11458333		780.631579	32
99000	2672389432	237	99	2.39393939		805.026316	33
102000	2453694535	398	102	3.90196078		829.421053	34
105000	2252896527	419	105	3.99047619		853.815789	35
108000	2068530817	362	108	3.35185185		878.210526	36
111000	1899252669	474	111	4.27027027		902.605263	37
114000	1743827392	927	114	8.13157895		927	38
117000	1601121337	516	117	4.41025641		951.394737	39
120000	1470093628	705	120	5.875		975.789474	40
123000	1349788566	736	123	5.98373984		1000.18421	41
126000	1239328665	493	126	3.91269841		1024.57895	42
129000	1137908247	578	129	4.48062016		1048.97368	43
132000	1044787565	862	132	6.53030303		1073.36842	44
135000	9592874106	665	135	4.92592593		1097.76316	45
138000	8807841583	874	138	6.33333333		1122.15789	46
141000	8087052171	1052	141	7.46099291		1146.55263	47
144000	7425248535	870	144	6.04166667		1170.94737	48
147000	6817603576	1052	147	7.15646259		1195.34211	49
150000	6259685222	757	150	5.04666667		1219.73684	50

Through the use of a Java program, the recursive implementation of the n^{th} Fibonacci Number calculation was shown to run in $O(2^n)$ time, while the dynamic programming implementation ran in O(n) time. This reinforces

the stated argument that the recursive implementation is unsuitable for calculating large values of n and that the dynamic programming implementation is better for any purpose. The next section will discuss the details of how the program used for this experiment was created.

Programming Implementation

Coding of the n^{th} Fibonacci Number calculator was done in Java, with both versions of the algorithm contained as methods within a single class, Nth_Fib. A helper method, run_algo(), was created to select the version of the algorithm to run, track experimental and theoretical runtimes, calculate c, and manage input and output. No dedicated classes or frameworks were used to record runtimes; the system time was taken before and after each run and the difference was the runtime. Because output data could be very large, the console was only used for monitoring progress, while experimental results were saved in a .csv file. Furthermore, the calculated values of F_n grew extremely large, so the Long and BigInteger data types were used. For example, in the table above for $F_{3,000}$ to $F_{150,000}$, the values of F_n are thousands of digits long, but are not fully shown in the cells. Because of this, the dynamic programming implementation also reaches a limit based on hardware memory availability. In fact, the machine running the tests ran out of memory around $F_{180,000}$. A more space-efficient implementation is possible, but for simplicity was not used here. This program also has no user interface; all experimental controls are handled through the main method in the Nth_Fib class by calling the helper method with the parameters to be investigated. To run the program with the experimental configuration in this report, the nth_fib.jar file can be executed from the command line or from a Windows/Mac/Linux GUI.

Additional output formatting, calculation of c_{min}, c_{max}, and related bounds, and graphing was done in Excel. With program output being stored in .csv files, these steps only required using Excel's included tools and converting the document to .xlsx format. Since multiple files were created, raw output was cut and pasted into a single file.

Conclusions

The primary observation of this experiment is that even simple algorithms can become impossibly slow, and that efficient algorithms need not be complicated. The recursive method of calculating the n^{th} Fibonacci Number is easy to understand and program, but useless beyond n = 50 because of runtime growth. With dynamic programming, this problem is solved and without adding more than a few lines of code to the already brief program. Despite their simplicity, the Fibonacci Numbers and their generating function play an important role in mathematics and science. Research into and observations of these numbers has gone on for centuries and will continue as long as humans search for patterns in the world around them.

Sources

"Bee Ancestry." University Child Development School (2007): n. pag. Web. <http://www.ucds.org/spark/magazine-curriculum/Fibonacci_BeeAncestry.pdf>.

Cormen, Thomas H., Charles Eric. Leiserson, Ronald L. Rivest, and Clifford Stein. "Fibonacci Heaps." Introduction to Algorithms. Third ed. Cambridge (Mass.): MIT, 2009. 506-30. Print.

"Fibonacci Heap." Growing with the Web. N.p., 19 June 2015. Web. 20 Mar. 2016. <http://www.growingwiththeweb.com/2014/06/fibonacci-heap.html>.

"The Fibonacci Quarterly." The Fibonacci Quarterly. Ed. Curtis Cooper. N.p., n.d. Web. 07 Mar. 2016. <http://www.fq.math.ca/index.html>.

"Golden Ratio." Wikipedia. Wikimedia Foundation, n.d. Web. 20 Mar. 2016. <https://en.wikipedia.org/wiki/Golden_ratio>.
Knott, R. "The Life and Numbers of Fibonacci." The Life and Numbers of Fibonacci. +Plus Magazine, n.d. Web. 20 Mar. 2016. <https://plus.maths.org/content/life-and-numbers-fibonacci>.
Knuth, Donald Ervin. The Art of Computer Programming. Fundamentals Algorithms. 3rd ed. N.p.: n.p., 1968. 100. Print.
Mycodeschool. "Time Complexity Analysis of Recursion - Fibonacci Sequence." YouTube. YouTube, 10 Oct. 2012. Web. 22 Mar. 2016. <https://www.youtube.com/watch?v=pqivnzmSbq4>.