Estimate Secret Algorithms #2

Jonathan Wenger

Client Code Summary

The way I made my client code is I went through each of the four algorithms starting at a small-ish set of 500 elements, and then doubling until the program took way too long to load (2+ hours for one algorithm was my limit for that algorithm). I tested each algorithm a given number of times at each given amount of data (the faster the algorithm took, the more times I tested it), and took the average of those times for precision purposes. Some algorithms were able to run all the way to the breaking point of roughly 524,288,000, but others had to stop at much lower values.

My client code had 3 methods: main, goThroughAlg, and runAndClockAlg, and they each called each other. Meaning, main calls goThroughAlg, and goThroughAlg calls runAndClockAlg.

Before I wrote the official code, I played around with the algorithms to see how long they would take, and see the highest amount of data that could be used in that algorithm without my machine taking forever, and I found the data breaking points. For algorithm 1, it was roughly 33,000 pieces of data, for algorithm 2 it was 524,288,001, for algorithm 3 it was 2,049,000, and for algorithm 4 it was 524,288,001. I also figured out how many times to run the tests on each algorithm at each data size; the faster the

algorithm took, the more times I wanted to test it for precision purposes. For algorithm 1, I tested it 3 times (very slow algorithm), for algorithm 2 I tested it 1,000 times (very fast algorithm), for algorithm 3 I tested it 5 times (also slow algorithm), and for algorithm 4 I tested it 10,000 times (super super fast algorithm).

My main method called goThroughAlg on each algorithm, which started each algorithm at 500 pieces of data, constantly doubling all the way up to the data breaking points (as mentioned in the previous paragraph).

goThroughAlg is a method that tests the given algorithm at all the different data input sizes, starting at 500 and doubling all the way to the data breaking point that I found earlier. This method tested the algorithm at each data size multiple times (depending on how fast the algorithm ran, the more times I tested it), and this method took the average of those times to try to be as accurate as possible. This method took in 3 different parameters: the algorithm (either 1, 2, 3, or 4), the data breaking point (that I figured out by testing earlier), and the amount of times to test at each data size. (the faster the algorithm, the more times I tested). This method first printed out that we are starting to test this algorithm (on line 15 of the code). Then, it used a for loop, starting at 500 going all the way to the data breaking point, and doubling at every iteration of the loop. Inside the loop, I created a long data type variable called “time” and set it equal to 0 (line 19). This variable will end up being the average time it took for that algorithm to run with that amount of data inputted. I then created an inner for

loop, going from 0 to the amount of times to test (the third parameter of this method), calling runAndClockAlg (which ran the algorithm at that given data size the outer for loop variable was equal to), and added what was returned from runAndClockAlg (the amount of time (in nanoseconds) it took to run that algorithm) to the “time” variable (runAndClockAlg will be explained in more detail in the next paragraph). I then broke out of the inner loop. I then divided “time” by the number of times the test was run (third parameter of the method), which got me the average of the iterations of that algorithm with that amount of data. I then printed out that average.

runAndClockAlg is a method that took in 2 parameters: the algorithm, and the amount of data that we are testing at this iteration (n). This method starts by calling setup() on the algorithm, using the amount of data given as a parameter (line 34). Then, it takes the current nanotime by calling System.nanoTime(), and saving it as a variable called timeBefore (which is the time before the algorithm runs). Then, I call execute() on the algorithm. After, I found the time (in nanoseconds) it took to run that algorithm by subtracting timeBefore from System.nanoTime(), and saved that amount of nanoseconds into a variable called timeItTook. This method returned timeItTook, so that we would be able to find the average time it took in the goThroughAlg method, as explained in the previous paragraph.

This program printed the algorithm average time of all of the attempts at each data size, and it then doubled the data set. This pattern repeated until the data set

reached its maximum, and then it went onto the next algorithm. With all of that data printed, I was now able to see how long each algorithm took at each data size. I then put the N (the amount of data) and the F(N) (the average time of the attempts, in nanoseconds) into a Microsoft Excel spreadsheet to put into a table and to graph.

Here is a picture with some (NOT ALL) of the sample output (NOT MY ACTUAL DATA):

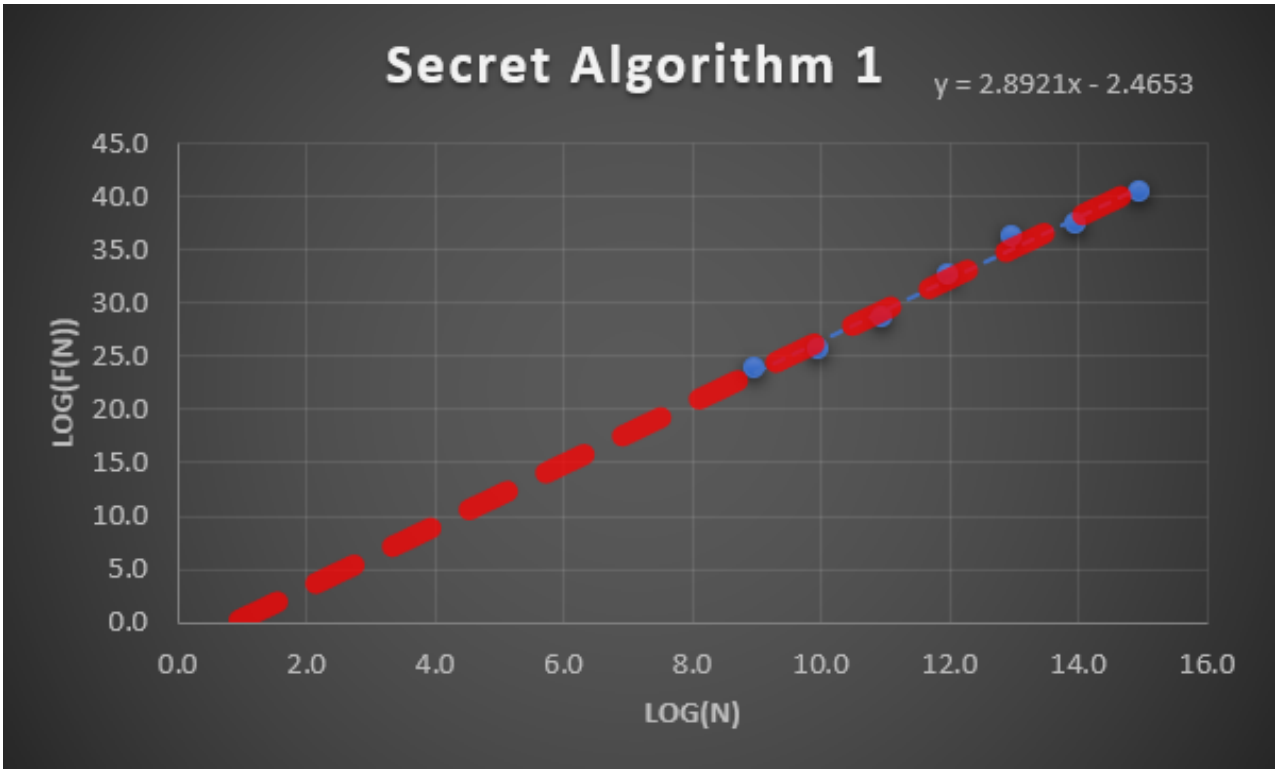
```
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm2 with 128000 pieces of data is 32260 nanoseconds
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm2 with 256000 pieces of data is 61122 nanoseconds
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm2 with 512000 pieces of data is 112403 nanoseconds
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm2 with 1024000 pieces of data is 267848 nanoseconds
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm2 with 2048000 pieces of data is 623587 nanoseconds
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm2 with 4096000 pieces of data is 1315573 nanoseconds
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm2 with 8192000 pieces of data is 2643421 nanoseconds
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm2 with 16384000 pieces of data is 5178059 nanoseconds
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm2 with 32768000 pieces of data is 10423563 nanoseconds
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm2 with 65536000 pieces of data is 22384020 nanoseconds
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm2 with 131072000 pieces of data is 45315592 nanoseconds
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm2 with 262144000 pieces of data is 89867865 nanoseconds
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm2 with 524288000 pieces of data is 170625286 nanoseconds
Starting algorithm class edu.yu.introtoalgs.SecretAlgorithm3
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm3 with 500 pieces of data is 1276840 nanoseconds
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm3 with 1000 pieces of data is 1090000 nanoseconds
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm3 with 2000 pieces of data is 792380 nanoseconds
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm3 with 4000 pieces of data is 2105800 nanoseconds
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm3 with 8000 pieces of data is 9333380 nanoseconds
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm3 with 16000 pieces of data is 33440240 nanoseconds
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm3 with 32000 pieces of data is 129700319 nanoseconds
Average it took to run class edu.yu.introtoalgs.SecretAlgorithm3 with 64000 pieces of data is 557087720 nanoseconds
```

Graphing Note

Just to note, for the four log-log graphs I'm about to show (one per algorithm), the blue dots in the back of the graph are the actual points from the log-log table, and the red line is the line of best fit (generated by Microsoft Excel). Additionally, in the top right corner, I have the equation of that line of best fit (also generated by Excel), which I use in my reasoning for determining the order of growth of these algorithms,

Secret Algorithm #1

N (Amount of Data)	F(N) (Nanoseconds)	Log(N) - Base 2	Log(F(N)) - Base 2
500	15164200.0	9	23.9
1000	55456025.5	10	25.7
2000	395270125.5	11	28.6
4000	7117419500.0	12	32.7
8000	76766599550.0	13	36.2
16000	192914528525.5	14	37.5
32000	1523140867900.0	15	40.5

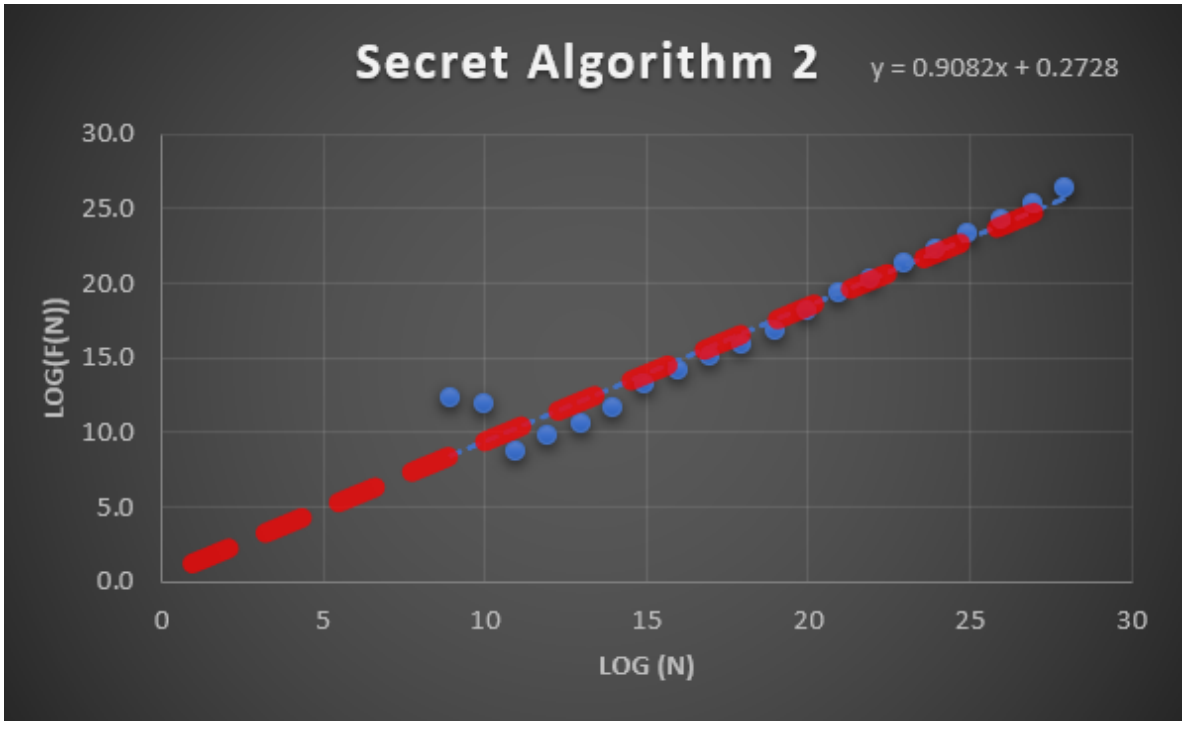


I believe the order of growth of this algorithm is cubic. The slope of the log-log graph is 2.8921, which is extremely close to 3, and we learned that if a log-log graph is linear, the slope of the log-log graph determines the exponent of the original function aN^b, where the slope of the line is equal to b. Because the slope of this log-log graph is

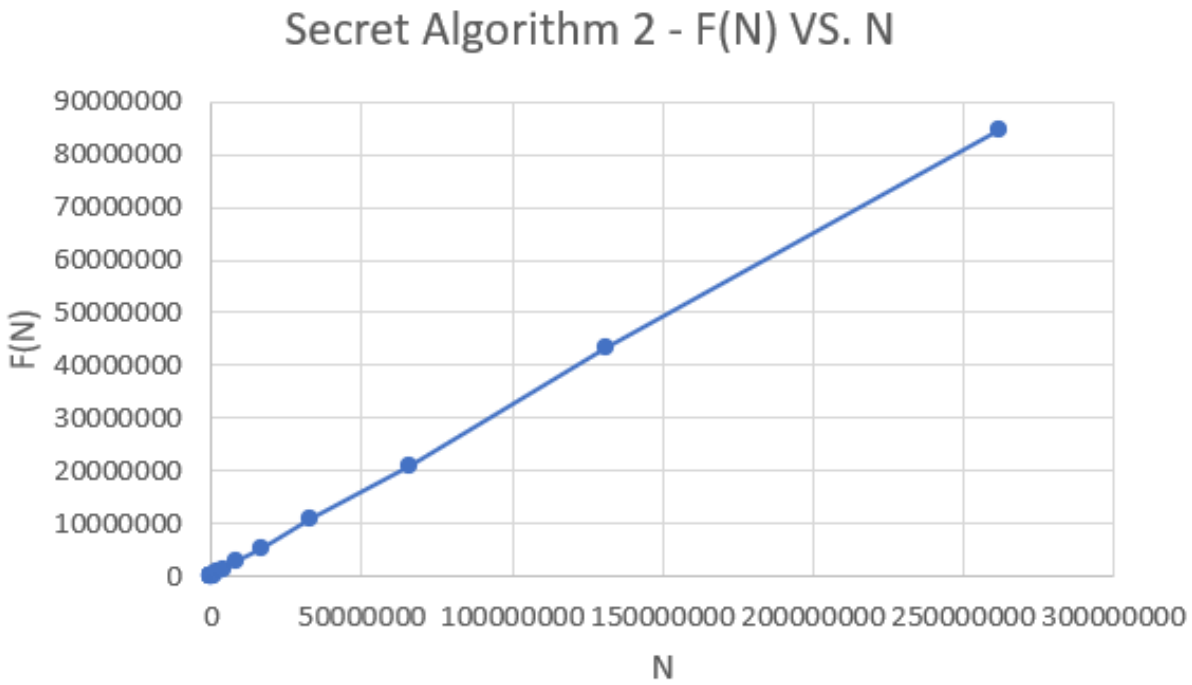
extremely close to 3, the function would be close to aN^3, In that case, I believe it is safe to say that this algorithm's order of growth is cubic.

Secret Algorithm #2

N (Amount of Data)	F(N) (Nanoseconds)	Log(N) - Base 2	Log(F(N)) - Base 2
500	4968.2	9	12.3
1000	3750.0	10	11.9
2000	438.4	11	8.8
4000	904.5	12	9.8
8000	1481.5	13	10.5
16000	3155.9	14	11.6
32000	9572.2	15	13.2
64000	18469.0	16	14.2
128000	34045.3	17	15.1
256000	61396.0	18	15.9
512000	119304.1	19	16.9
1024000	279824.2	20	18.1
2048000	640800.9	21	19.3
4096000	1313227.3	22	20.3
8192000	2655285.8	23	21.3
16384000	5258088.7	24	22.3
32768000	10652124.3	25	23.3
65536000	20798793.5	26	24.3
131072000	43410811.6	27	25.4
262144000	84588309.3	28	26.3

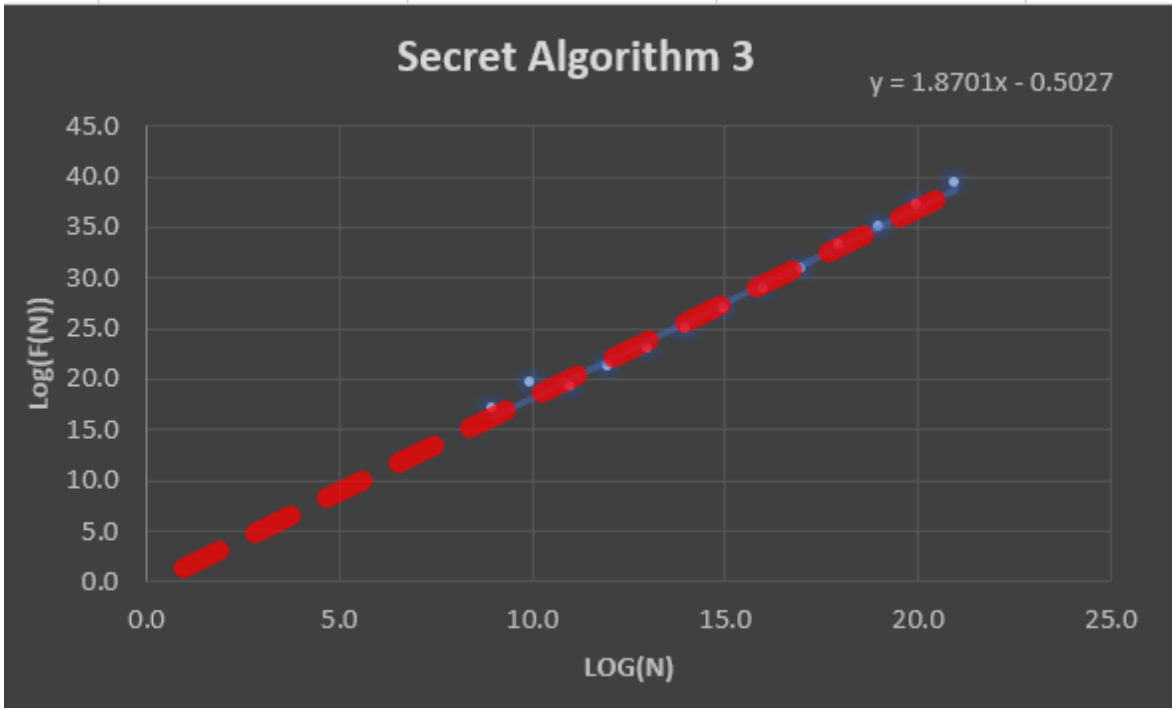


I believe that the order of growth for this algorithm is linear. Firstly, the slope of the log-log graph is .9082, which is very close to 1. As we mentioned earlier, that means the function of the original graph (not the log-log) is close to aN^1, which is the same as aN, which is a linear function. Additionally, when I remove the first four points from the graph (or lines from the table), the slope of the line on the log-log graph then becomes 1.0439, which is even closer to 1. I believe that the reason why those four points are farther away from the line than the other points and make the slope farther away from 1 is because the program ran so quickly with such small amounts of data that those data points on the table and graph are more "unreliable" than other points. Finally, the reason why I believe the order of growth of this algorithm is linear is because when I look at the graph of F(N) Vs. N (NOT the log-log graph, but the original graph), the graph is also basically linear, which shows that there is a linear relationship between the two (see graph below). These are the reasons why I believe the order of growth of this algorithm is linear.



Secret Algorithm 3

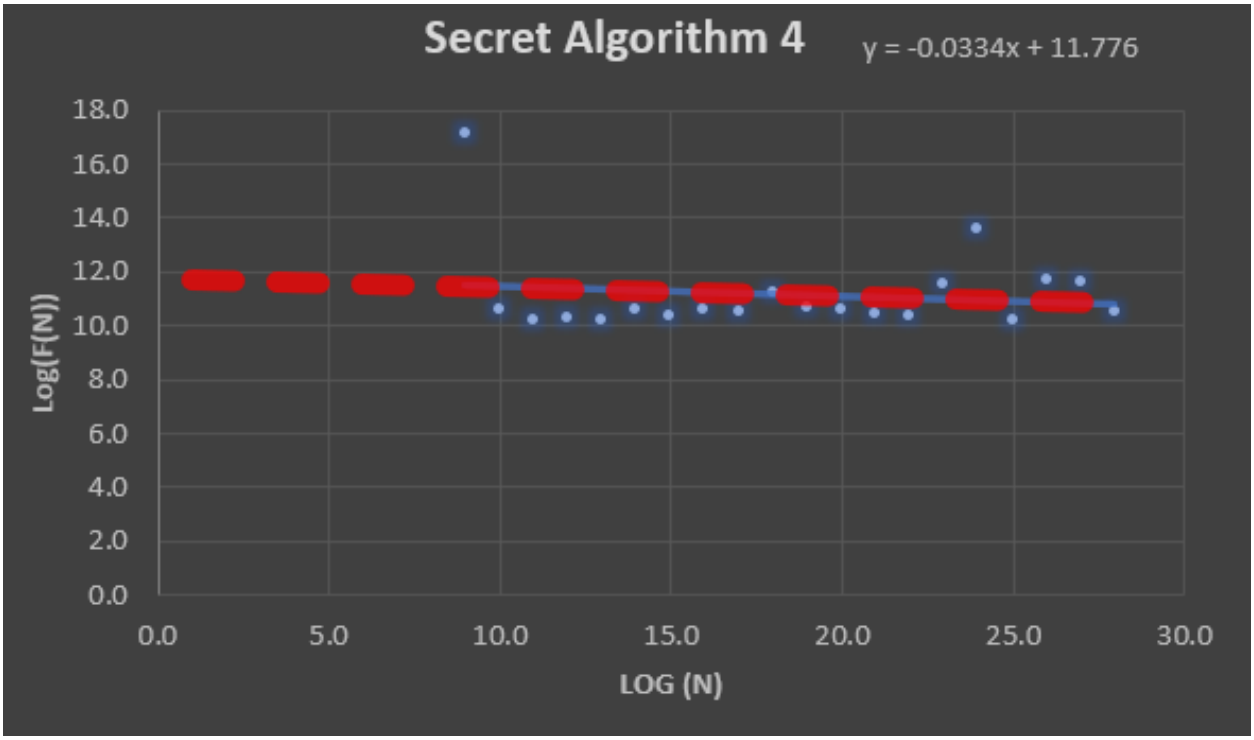
N (Amount of Data)	F(N) (Nanoseconds)	Log(N) - Base 2	Log(F(N)) - Base 2
500	134519.3	9	17.0
1000	891660.0	10	19.8
2000	617100.0	11	19.2
4000	2562440.0	12	21.3
8000	9012580.0	13	23.1
16000	33743720.0	14	25.0
32000	128896280.0	15	26.9
64000	520762999.3	16	29.0
128000	2070585400.0	17	30.9
256000	11042652519.3	18	33.4
512000	37254011200.0	19	35.1
1024000	157436737040.0	20	37.2
2048000	696784448240.0	21	39.3



I believe the order of growth for this algorithm is quadratic, as the slope of the log-log graph is 1.8701, which is very close to 2. As we mentioned earlier, this means the original function is close to aN^2, which shows that this algorithm is quadratic.

Secret Algorithm 4

N (Amount of Data)	F(N) (Nanoseconds)	Log(N) - Base 2	Log(F(N)) - Base 2
500	144220.0	9	17.1
1000	1600.0	10	10.6
2000	1200.0	11	10.2
4000	1260.0	12	10.3
8000	1180.0	13	10.2
16000	1600.0	14	10.6
32000	1300.0	15	10.3
64000	1560.0	16	10.6
128000	1460.0	17	10.5
256000	2480.0	18	11.3
512000	1620.0	19	10.7
1024000	1600.0	20	10.6
2048000	1400.0	21	10.5
4096000	1340.0	22	10.4
8192000	2960.0	23	11.5
16384000	12200.0	24	13.6
32768000	1200.0	25	10.2
65536000	3340.0	26	11.7
131072000	3100.0	27	11.6
262144000	1440.0	28	10.5



I believe that the order of growth for this algorithm is constant time. Firstly, the slope of this log-log graph is extremely close to 0, which would mean the original function is equal to aN^0, which equals a, which is a constant. There is, however, one point that is an outlier in this graph, which is the first line in the table, in which the algorithm seemingly took a lot more time than usual with such a small data set. I would like to suggest why this is the case; because that was the first data set to run this algorithm, it probably took a slow time to start up, which wasn't the case with the other sized data sets, as the algorithm had already been started already. Another reason why I believe this function is constant time is because if you look at the table, specifically the values of N and F(N), you notice that regardless of the size of N, all of the values of F(N) are extremely close to each other (remember, this F(N) is in nanoseconds, so even though they're thousands of numbers apart from each other, it's still talking about the tiniest of values). Because there isn't constant growth, as the numbers of F(N) fluctuate between roughly 1,000 and 3,500 (with two outliers) without any pattern, I believe this function is constant time. However, it is also possible to say that this algorithm is logarithmic (as logarithmic algorithms are very very close to constant time algorithms' run speed, and the slope of an algorithmic log-log graph would also be 0,) but because there isn't constant growth as I mentioned earlier, I don't believe this algorithm is logarithmic, as logarithmic functions have constant growth

while constant time functions do not. Therefore, I believe that the order of growth for this algorithm is constant time.