Isaac Gutt

Introduction to Algorithms

Estimate Secret Algorithms #2

My client code consists of two classes, one in the src\main\java directory, and one in the src\test\java directory.

The main class, EstimateSecretAlgorithms2.java, contains one inner class named Stopwatch, and a public method named timeAlg.

Stopwatch is a similar implementation from Sedgewick's book with an added Boolean parameter: Since SecretAlgorithm4 was too fast for milliseconds, I changed the long variables in the class to be equal to System.nanoTime instead of System.currentTimeMillis if the the boolean parameter is equal to true. If the boolean is false (algorithm 1-3), then the long variables are in milliseconds. The contructor just sets a long variable start to System.nanoTime or System.currentTimeMillis when the constructor is called, and when the elapsedTime method is called it takes the current time and returns (current – start).

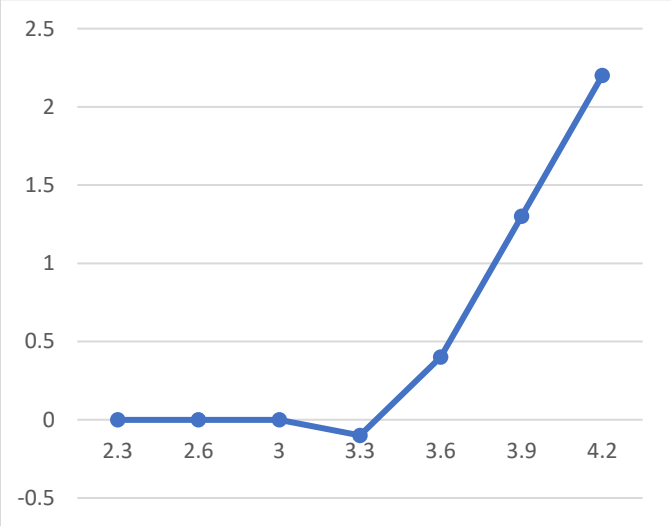
TimeAlg takes two parameters, the integer n to pass into the algorithm, and the algorithm instance itself. It calls setup(n), and instantiates the stopwatch and passes in the Boolean, which just checks if the algorithm is an instance of SecretAlgorithm4. After, it calls execute(), and when that's done the method returns elapsedTime from the stopwatch.

In the test directory, I have a class called TimerTest. In it I have 4 tests, one for each algorithm. I did it this way because I like being able to run one at a time with the run test buttons next to each test on Intelij. Each test has an instance of the algorithm specified by its title (alg1 has a SecretAlgorithm1 instance and so on). After, each test has a loop that starts with a different number based on how much each algorithm can handle, and it doubles up

until the loop stops which is also a different limit for each. Inside the loop I call the timeAlg method, passing in n and the algorithm, and I print n, the result of timeAlg which is f(n) and the ratio. I keep a double variable named previous which keeps track of the last f(n) of the algorithm, so when I divide the current calculation of timeAlg with the previous one, I get a ratio telling me how much longer it took to run the algorithm with the current n value than the last one.

Not all the algorithms were timed with the same measurements, so that will be explained in the algorithm sections.

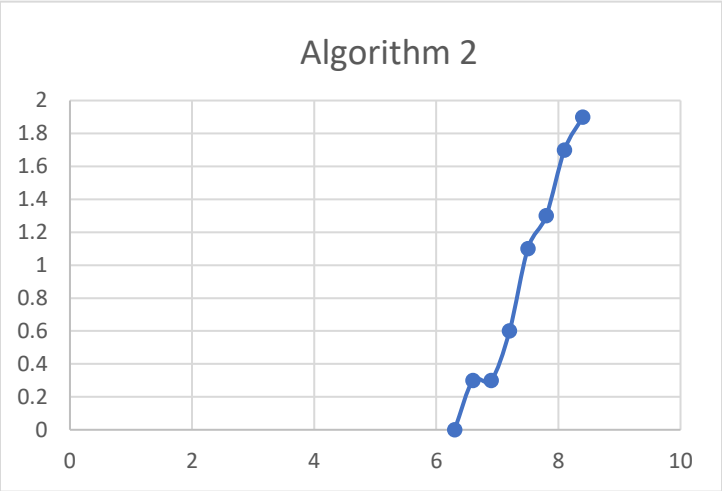
Algorithm 1 (timed with seconds):



n	log n	f(n)	log(f(n))	ratio
250	2.3	0	0	
500	2.6	0.011	0	
1000	3	0.035	0	3.18
2000	3.3	0.7	-0.1	20
4000	3.6	2.7	0.4	3.8
8000	3.9	21.9	1.3	8.1
16000	4.2	187.5	2.2	8.56

Algorithm 1 was very slow, so I was able to measure it in regular seconds. All I did was divide the result of timeAlg which returns milliseconds, by 1000. The average ratio is 8.7, and the ratio's constant rate at the end is near 8. Therefore, my decision is that the order of growth for this algorithm is N^3.

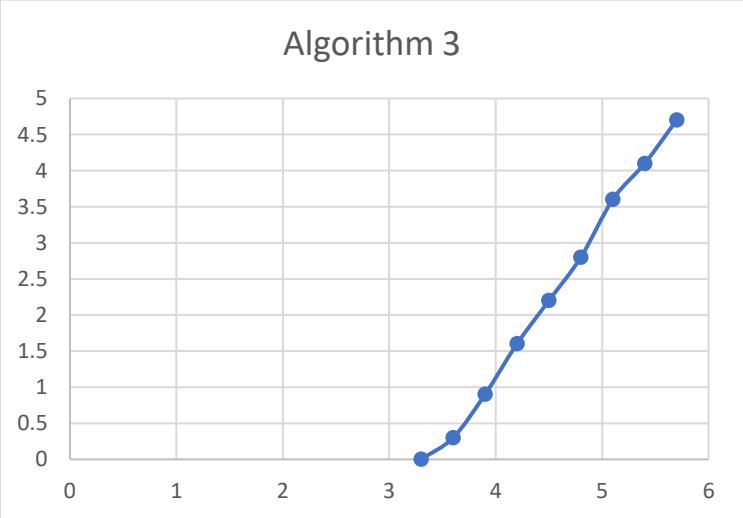
Algorithm 2 (timed with milliseconds):



n	log n	f(n)	log(f(n))	ratio
2048000	6.3	1	0	
4096000	6.6	2	0.3	2
8192000	6.9	2	0.3	1
16384000	7.2	5	0.6	2.5
32768000	7.5	13	1.1	2.6
65536000	7.8	20	1.3	1.5
131072000	8.1	51	1.7	2.5
262144000	8.4	91	1.9	1.8

Algorithm 2 was very fast, so I was able to time with extremely high values and used milliseconds (unchanged from the timeAlg method). The average ratio is 1.98, and the ratio's constant is around 2, so my decision is that the function's order of growth is linearO(n).

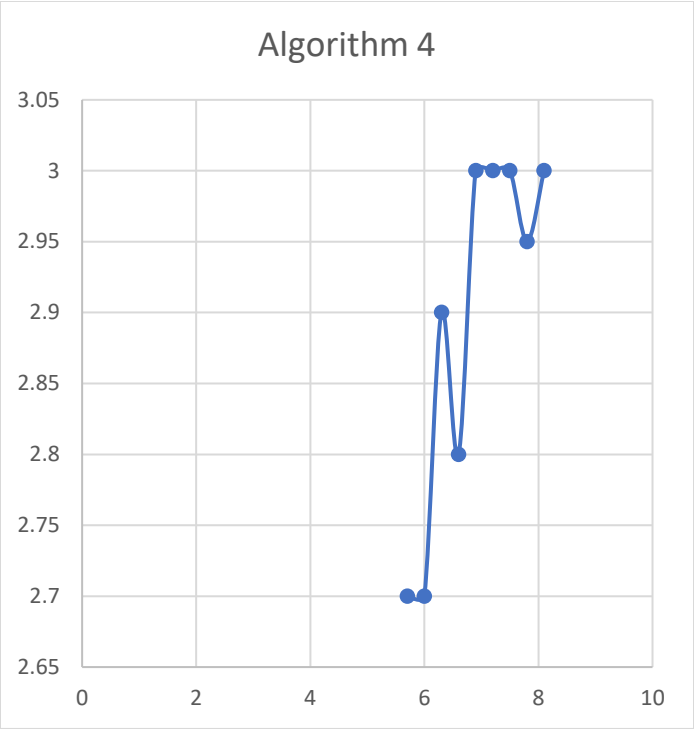
Algorithm 3 (timed with milliseconds):



n	log n	f(n)	log(f(n))	ratio
2000	3.3	1	0	
4000	3.6	2	0.3	2
8000	3.9	8	0.9	4
16000	4.2	45	1.6	5.6
32000	4.5	182	2.2	4.04
64000	4.8	698	2.8	3.83
128000	5.1	4363	3.6	6.2
256000	5.4	14074	4.1	3.225
512000	5.7	56155	4.7	3.98

While not as slow as algorithm 1, algorithm 3 got consistently far slower as n grew. The average ratio is 3.6, and the constant ratio was near 4, so my decision is that the functions order of growth is O(n^2).

Algorithm 4 (timed with nanoseconds):



n	log n	f(n)	log(f(n))	ratio
512000	5.7	600	2.7	
1024000	6	600	2.7	1
2048000	6.3	800	2.9	1.3
4096000	6.6	700	2.8	0.8
8192000	6.9	1000	3	1.4
16384000	7.2	1000	3	1
32768000	7.5	1000	3	1
65536000	7.8	900	2.95	0.9
131072000	8.1	1000	3	1.1
262144000	8.4	800	2.9	0.8
524288000	8.7	900	2.95	1.1

Algorithm 4 was the fastest, and the average ratio was close to 1, and n's size barely affects the speed as it grows so my decision is that the algorithm is log(n).