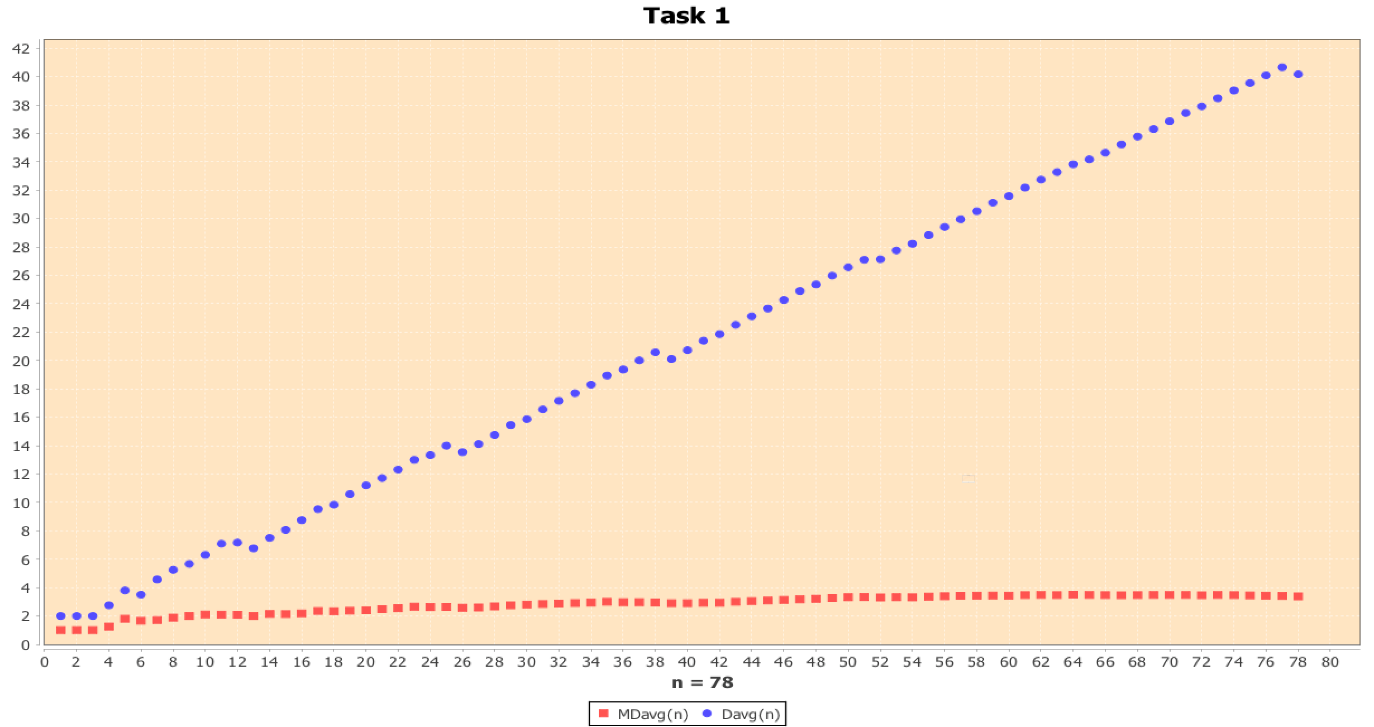
CS415 Project 1 Report

Task 1 Report: 

In the scatterplot above, in blue we have the average number of Divisions, Davg(n), for the Consecutive Integer Checking method, and in red we have the average number of Modulo Divisions, MDavg(n), done by Euclid’s algorithm. On the x-axis we measure the input, **n**, and on the y-axis we measure the average number of divisions done. The values for each axis in this graph go from 1 – 78. The values of **n** plotted for each algorithm was each integer in the 1-78 range.

Graph Analysis:

While the values of the two axes are relatively small in the grand scheme of things, it’s big enough to see an immediate difference between the two algorithms. For the Consecutive Integer Checking algorithm we can see the plots of its values aggregate around a straight line, demonstrating a constant growth as the value of **n** increases. At n = 1 the algorithm makes 2 divisions on average and at n = 78 the algorithm makes on average around 40 divisions. On the other hand, Euclid’s Algorithm’s average number of modulo divisions barely increases from the first n value (1) where the number of average divisions = 1, to the last n value measured (78), where the average number of divisions is roughly under 4. From the graph we can see the plots from Euclid’s form a line that has minimal, if any, growth, that goes straight across the graph.

From the shape and values of the Consecutive Integer Checking algorithm scatterplot we can determine that the algorithm’s average-case efficiency is most likely in **Θ (n)**, due to the linear growth of the divisions with the increase of n and the straight line that the plots form.

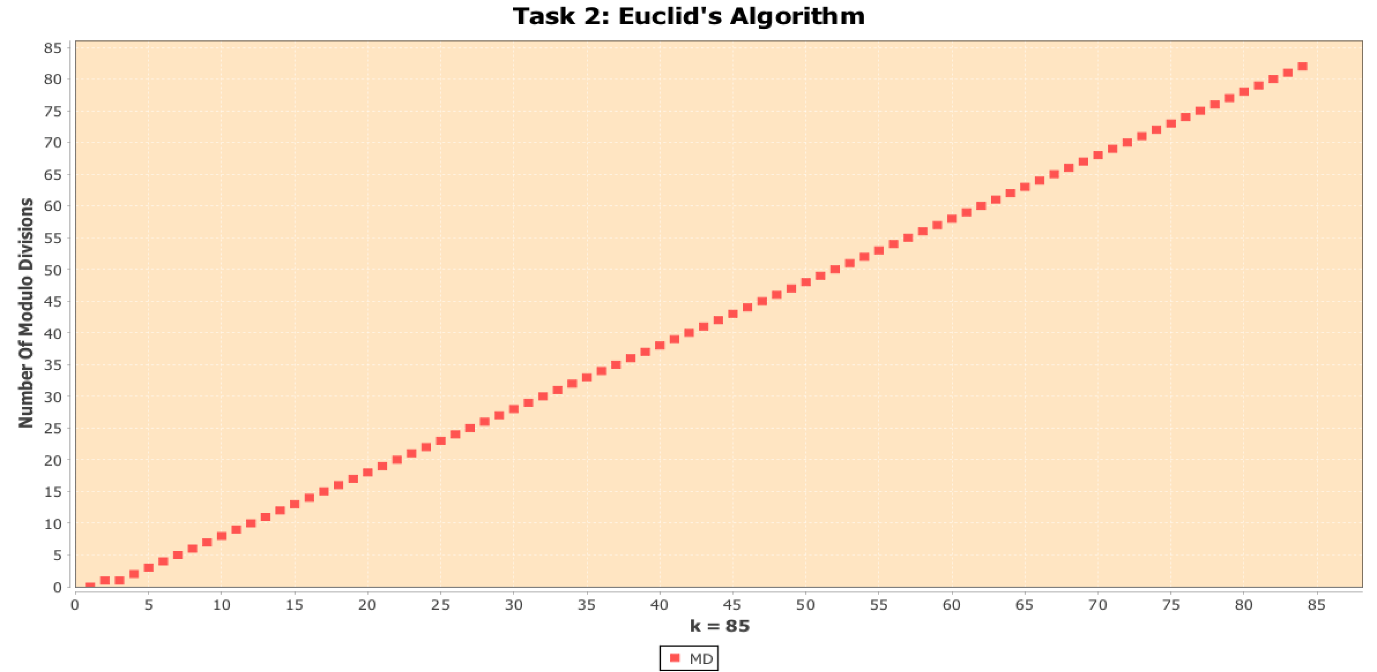
From the shape and values of the Euclid’s algorithm scatterplot we can determine that the algorithm’s average case-efficiency is most likely in **Θ (log n)**, due to how low the values are, though greater than 1.

*From this scatterplot we can extrapolate the values for the number of average divisions the two algorithms may take*, for values of n outside the range of the scatterplot. Below is a table with *predictions* made for every input, where n = 10k, for 1 ≤ k ≤ 5, as well the actual number of average divisions made by the algorithms, calculated by our program.

|  |  |  |
| --- | --- | --- |
| **Values of n** | **Predicted MDavg(n)** | **Predicted Davg(n)** |
| 10 | log 10 = 1 | n/2 = 5 |
| 100 | log 100 = 2 | n/2 = 50 |
| 1000 | log 1000 = 3 | n/2 = 500 |
| 10000 | log 10000 = 4 | n/2 = 5000 |

|  |  |  |
| --- | --- | --- |
| **Values of n** | **MDavg(n)** | **Davg(n)** |
| 10 | 1.8 | 5.5 |
| 100 | 3.57 | 51.75 |
| 1000 | 5.423 | 504.785 |
| 10000 | 7.3667 | 5009.3587 |

As you can see from the tables above, the predictions we made based off our scatterplot were fairly accurate in the larger scheme of values and so we can say confidently that the average case efficiency for Euclids is **Θ(log n)** and for consecutive integer checking **Θ(n).**

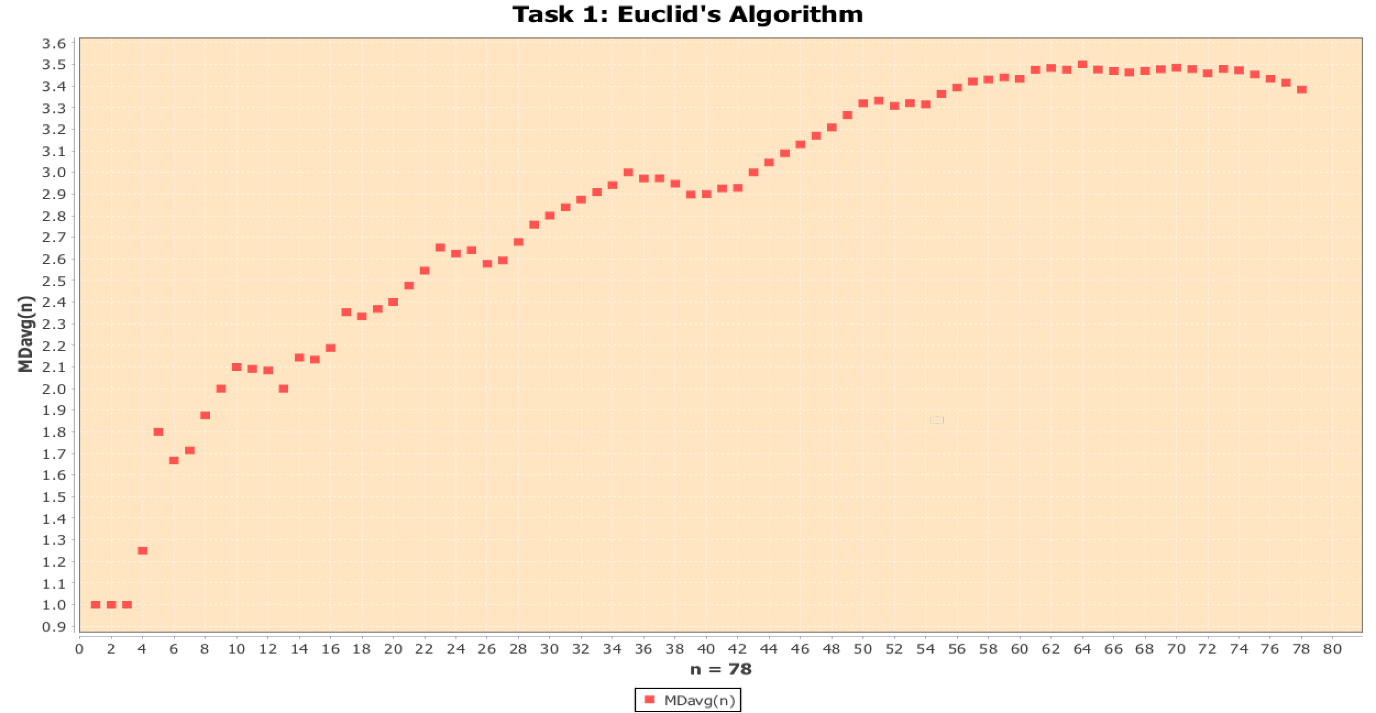
Task 2 Report:

The scatterplot above shows the number of modulo division done in Euclid’s for each value of k, where m and n in gcd(m, n) is f(k + 1) and f(k). The x-axis measures the value of k, going from 1-84, while the y-axis measures the number of modulo divisions done in gcd(f(k+1), f(k)).

Because the application was written in Java using the BigInteger data type to compute the nth Fibonacci number and GCD using Euclid’s algorithm, there is no upper bound for k demonstrated in this case because BigInteger has no cap on its max size (as large as the RAM on the computer can hold).

**However, if the application were to use a primitive 32-bit signed binary integers to compute the nth Fibonacci number then the upper bound for k would be no greater than 46. This is because that k = 46 computes the value 1836311903 and k = 47 computes the value 2971215073, which is greater than the maximum possible value for a 32-bit signed binary integer 2147483647.**

From the scatterplot we can see the plots form a straight, that increases at a linear rate. The number of modulo divisions done at each k is almost equal to k. From the graph we can infer that the worst case efficiency class for Euclid’s Algorithm is **Θ(n)**, judging from the linear nature the plots aggregate to on the graph. This is a significant difference from the complexity of the average case efficiency class of Euclid’s which we determined to be **Θ(log n)** in Task 1.

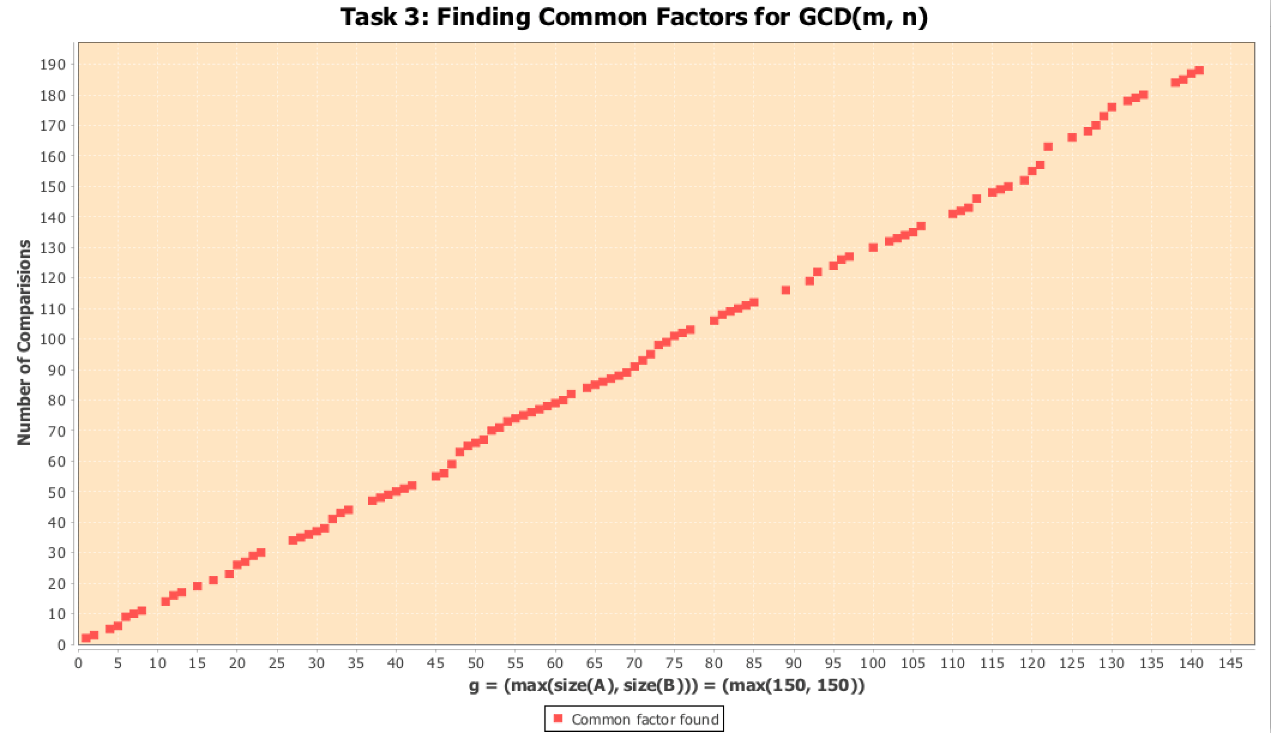
This is the scatterplot from Task 1 without the Consecutive Integer Checking average number of divisions. There is a large difference in the range of the y-axis between this graph and the worst-case graph above, as the worst case modulo divisions go to far greater values. To show this difference we can look at the value where n = 55 for Task 1 and compare it with the Task 2 graph where k = 11 (due to 55 being the 11th Fibonacci number in the sequence). The value of Task 2 is 11 which is already almost 2x more than any value in the Task 1 graph for Euclid’s algorithm. It’s larger than even the value we computed with our program for n = 105 for Task 1 which was 7.3667. Again, we can see that the worst case of Euclid’s algorithm is significantly less efficient then it’s average case.

In this task we can also compare the actual time it takes. The table below shows values computed by our program, where k = multiples of 10 ≤ 80.

|  |  |
| --- | --- |
| **Value of k** | **Time (in nanoseconds)** |
| 10 | 36305.9 |
| 20 | 68692.5 |
| 30 | 98772.0333 |
| 40 | 108993.8 |
| 50 | 123563.0667 |
| 60 | 134612.5 |
| 70 | 129432.02 |
| 80 | 189435.33 |

When measuring the time for this algorithm we can see that the time it takes to run increases as the value of k increases, at a steady rate. There is an exception at values 60 and 70 where 70 takes less time than k = 60. However, this seems more of an outlier as every other value increases at a linear rate, usually between 25000-30000 nanoseconds. This constant increase at a fairly fixed amount falls in line with our initial interpretation of the scatterplot in that the worst case efficiency Euclid’s algorithm falls into **Θ(n)**.

Task 3:



The scatter plot above shows the number of comparisons made when g = max(size(a), size(b)). The y-axis measures the number of comparisons while the x-axis is max(size(a), size(b)).

To exercise the efficiency of finding common factors for GCD(m, n) the input used to create the scatterplot are two lists A and B of max size 150 that contain a random number of small prime numbers generated at runtime. This is done to illustrate the scenario of finding common factors between two large integers.

From how the plots in the graph are placed, we can see that the plots form in a relatively straight line, which shows a linear growth. The number of comparisons is practically equal to max(size(a), size(b)) and as max(size(a), size(b)) = g we can see that the Middle-school procedure is in **Θ(g)**.