**Project 1 Report**

Skyler Penna, Darin Brown

Task 1:

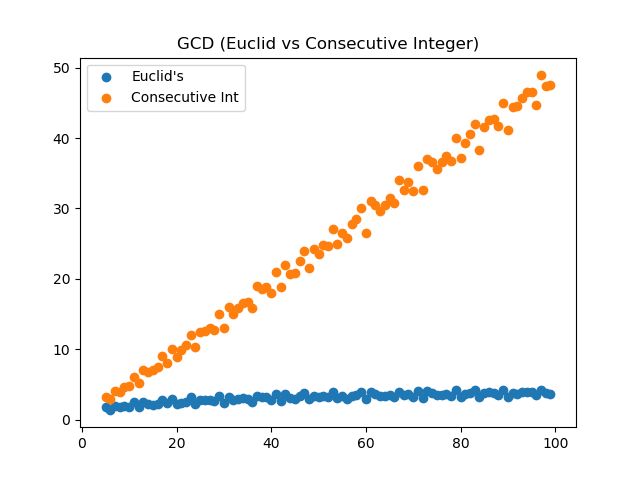
(x,y) = (inputs, operations)

Consecutive Integer:

From the graph below consecutive integer seems to have a slope of 1, which would imply that the algorithm’s efficiency is O(n).

Euclid’s Algorithm:

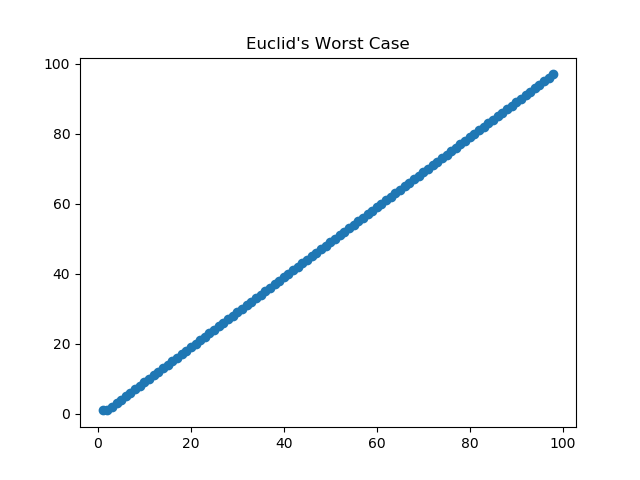
From the graph below Euclid’s algorithm is increasing at a slop that is less than 1. Because of this, it looks to be of (logn) in nature.



Task 2:

(x,y) = (inputs, operations)

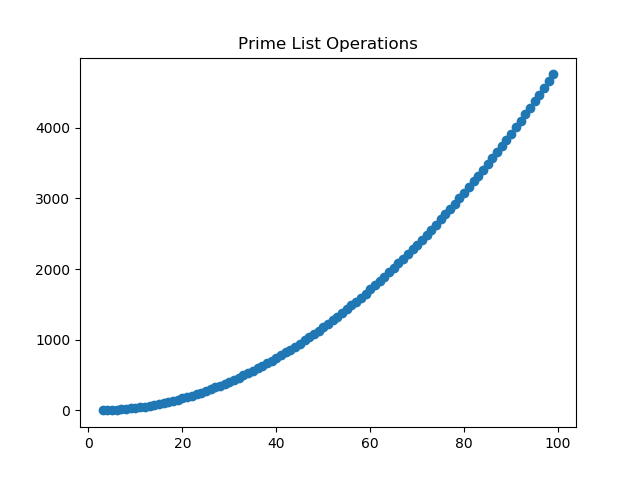
Euclid’s worst case (two consecutive Fibonacci numbers) has a slope of 1, which means its complexity is O(n). Compared to the average case, which was O(logn) it is significantly worse in efficiency. In fact Euclid’s worse case is of the same efficiency class as consecutive integer checking that was graphed above. While its operations are substantially larger than the average Euclid’s case the time taken for a machine to run the program is still slow, but it would still increase at a linear rate compared to Euclid’s.



Task 3:

Part1:

This algorithm to generate a list of primes definitely took longer than linear time complexity. When investigating the implementation of the algorithm we see that the algorithm must pass through two nested for loops in order to verify the numbers. This would give it a time complexity of O(n^2)



Part2:

This algorithm’s complexity is theta(n). This is achieved by creating two iterators i and j. If the ith element is larger than the jth element, j is incremented. If reversed, i is incremented, and finally if they are equal both I and j are incremented. This achieves linear theta complexity because it doesn’t have to use two for loops to traverse both lists but instead are traversing the lists at the same time based on which element is smaller. The graph isn’t a perfect line only because the lists generated are of random size and full of random elements which causes some variation.

