CS 415 Project 1

Keegan Donley, Jeff Hultman

0.1 Task 1

Maybe state if the values are unique or not?

Average-case efficiency of Euclid's algorithm and consecutive integer checking algorithm To test the average-case efficiency of these algorithms, we generate 100 random values of n from 1 to 70, then count the number of operations needed to calulate the average GCD for n using Euclid's algorithm and consecutive integer checking.

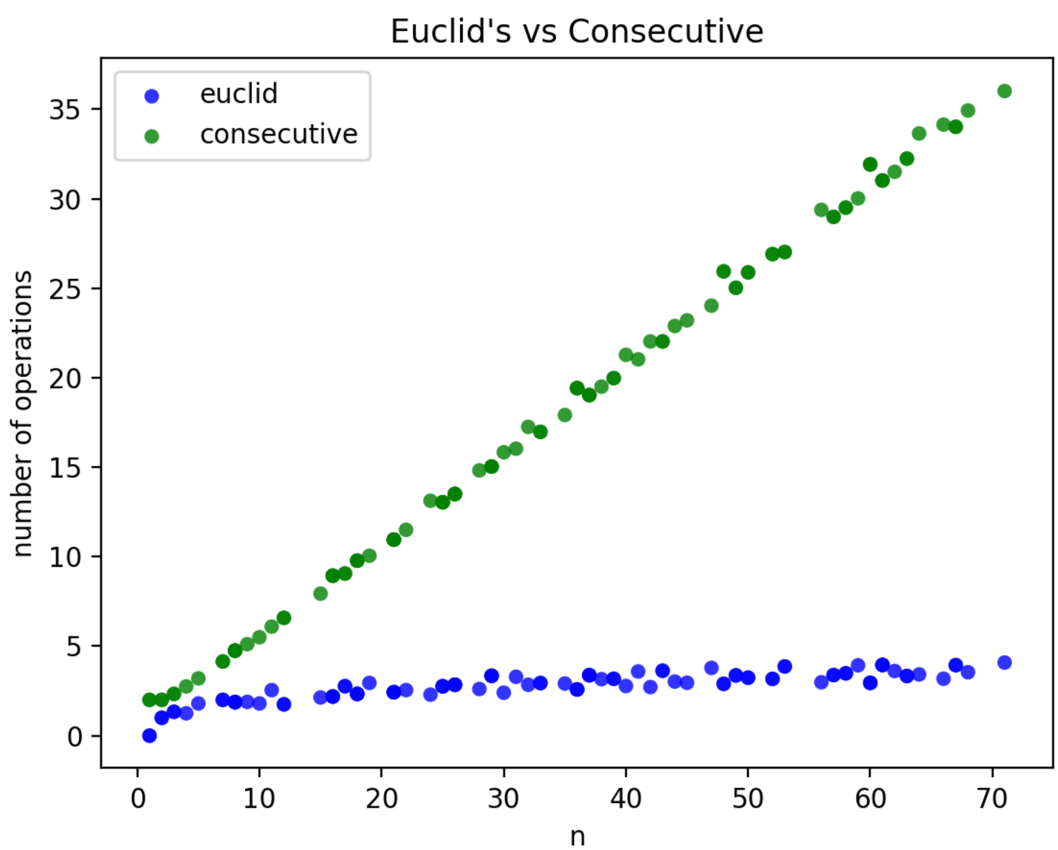
To calculate the average number of operations for n, we take the average number of operations for:

euclidGCD(n, 1), euclidGCD(n, 2), . . . , euclidGCD(n, n)

and

consecutiveGCD(n, 1), consecutiveGCD(n, 2), . . . , consecutiveGCD(n, n)

Go into more worded detail about how the algorithms compare. Since it's of the main points for the rubric for task 1



Euclid $\theta(\log n)$

In the average case, Euclid's algorithm runs in $\theta(\log n)$ time and took less than 5 modulo divisions.

Solid stuff!

Consecutive Integer θ(n)

The empirical testing shows the consecutive integer testing results to be nearly perfectly linear.

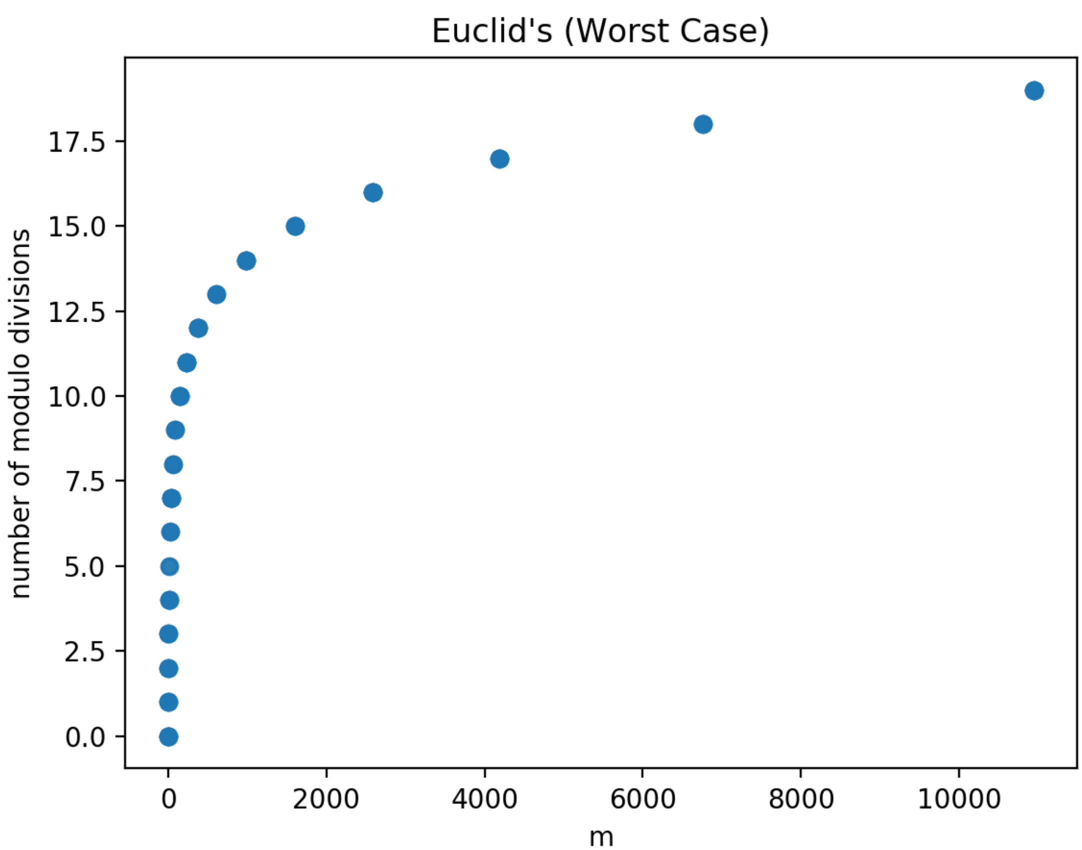
0.2 Task 2

Worst-case efficiency of Euclid's algorithm The worst case for Euclid's algorithm occurs when two consecutive integers from the Fibonacci sequence are used as m and n. To test the efficiency, we generate 200 values for k between 1 and 20. k is an index in the Fibonacci sequence, where m = k + 1 and n = k.

For example, based on the start of the Fibonacci sequence: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

k = 5 => gcd(8, 5)

k = 8 => gcd(34, 21)



Because the gcd of two consecutive Fibonacci numbers is always 1, the complexity in the worst case for Euclid's algorithm is θ(logn). We saw a trend that respresents a complexity of θ(logn) in the average case as well, however the number of modulo divisions tended to be lower than how many were needed in the worst case.

Time Analysis We analyzed the time taken by Euclid's algorithm in the average and worst case by testing 5000 values. For the average case, we use values from 0 - 4999, and for the worst case we use consecutive Fibonacci numbers leading up until position 5000 in the sequence.

Average Case: 0.00000109137s

Actual time or number of divisions? Maybe put both?

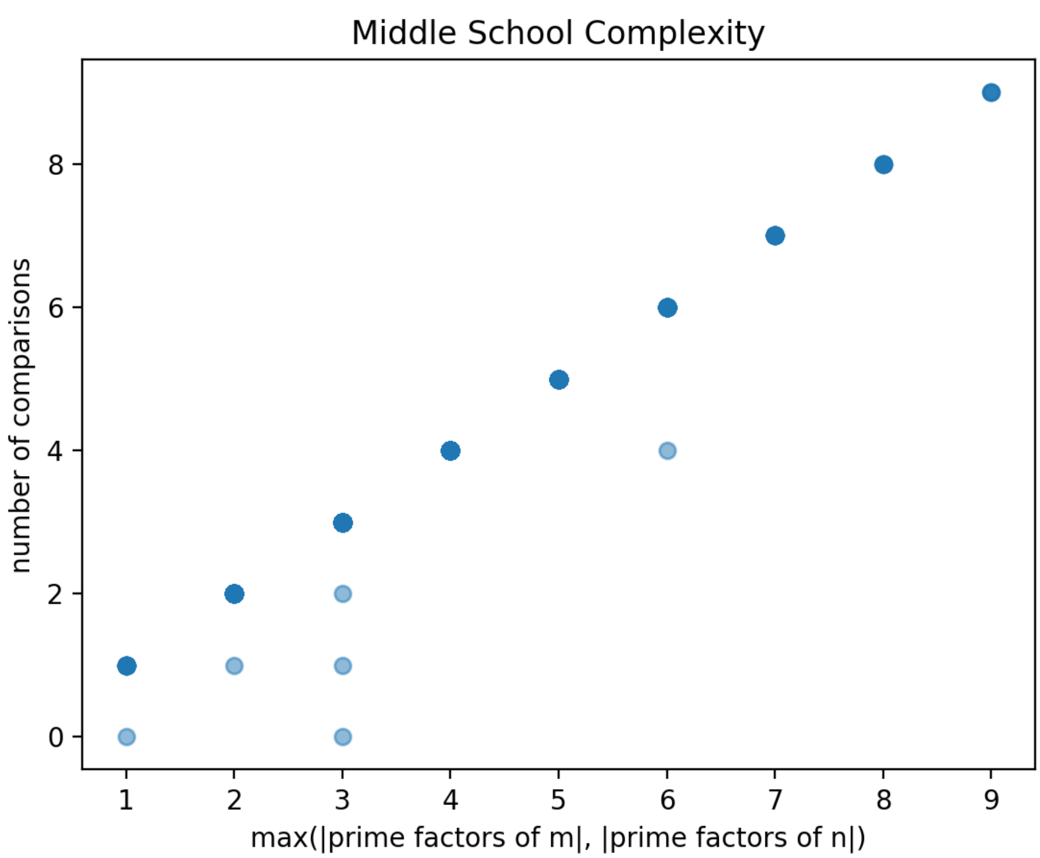
Worst Case: 0.001714296s

The average case ran 3 times faster than the worst case in our tests

Upper Bound of k We decided to set our upper bound of k to 1,000 because this is the point at which the algorithm starts to take a few seconds to run. Python can handle larger values, but for the sake of this empirical analysis larger values are unnecessary.

0.3 Task 3

The "middle-school procedure" To evaluate the complexity of the middle school method of calculating GCD, we generated 1000 pairs of random integers from 1 - 999 as m and n. When plotted, the results show that the middle school method is θ(n).



The evidence shows the complexity to be approximately linear, but there are some outliers where both m and n had prime factors, but none were common and the gcd was determined to be 1. Darker dots on the scatter plot indicate where the majority of data points lie.

A step through of
implementation and your
thoughts while implementing would be nice. There was a lot
work to create this algorithm
maybe talk about some details?