Jack Newman, John Salman
Dr.Gill, cs415, project 1
03/02/18

Project 1: report

Task 1:

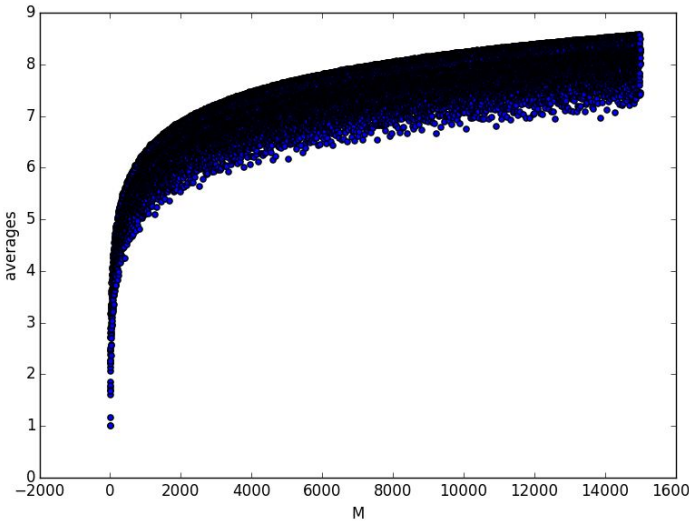
Within this section we were tasked to compare Euclid's Algorithm and the consecutive integer checking Algorithm for computing greatest common divisor (we will refer to this as gcd or gcd() within this report). All algorithms were written as functions in the format gcd(m,n) with m and n as positive integers with m >= n > 0.

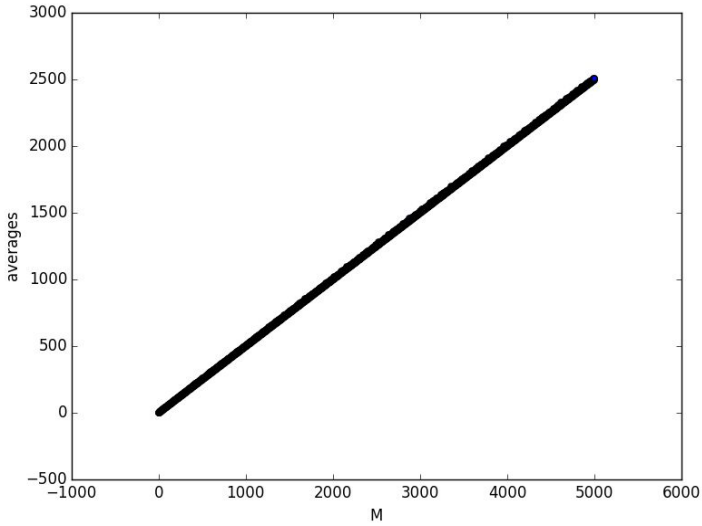
Euclid's Algorithm constitutes dividing m by n and assigning the remainder to an additional variable. This is repeated until the remainder is equal to 0, in which case you return the current value of m as the greatest common divisor of both m and n. Consecutive integer checking algorithm works by assigning n to an additional variable (we'll call this r) and checking if r divides both m and n with no remainder. If there is no remainder then r is the greatest common divisor, if this isn't the case we decrease the value of r and check again until it cleanly divides both m and n.

We analyzed both algorithms by calculating the average-case efficiency. This is found by averaging the number of modulo divisions for a given input n:

[gcd(n, 1) + gcd(n, 2) + ... + gcd(n,n)] / n

With Euclid's algorithm there is only one division per iteration, the current m divided by the current n. However, within the consecutive integer checking algorithm there are two divisions to check if the current variable is a common divisor of both m and n.





In the left diagram we see the average-case efficiency of Euclid's Algorithm on input sizes 0-15,000. We can tell from this scatterplot that the algorithm fall into $\Theta(logn)$ efficiency due to the curvature of the line. An interesting conjecture to be drawn from this graph is that as the

size of the input increases the line will start to approach a horizontal asymptote representing a maximum value for the average-case efficiency.

In the right diagram is the scatterplot average-case efficiency of the consecutive integer checking algorithm. The slope of the line clearly indicates a linear relationship between the average-case efficiency and the size of the input. This puts the algorithm into $\theta(n)$ efficiency class.

When comparing the two algorithms we now have definitive proof that Euclid's Algorithm is more efficient than the consecutive integer checking algorithm. This can be proven through the application of L' Hospital rule.

$$\lim_{n\to\infty} \frac{log_2 n}{n} \text{ -> } \lim_{n\to\infty} \frac{1}{nln(2)} \approx 0$$

A zero indicates the numerator grows slower than the denominator as the size of n increases. Which shows that $log_2 n \in O(n)$.

Task 2:

The next task had us implementing Euclid's algorithm on consecutive elements of the Fibonacci Sequence. To accomplish this task we wrote a program to generate a Fibonacci Sequence in C++. The program was an implementation of the algorithm to iteratively generate the sequence and place it into an array where F[i] = F[i - 1] + F[i - 2] for i > 1.

We found that due to the limitations of the language we could only generate a sequence of length 93. Our fibonacci(93) = 7540113804746346429, which can be proved as valid by adding fibonacci(92) + fibonacci(91), which is:

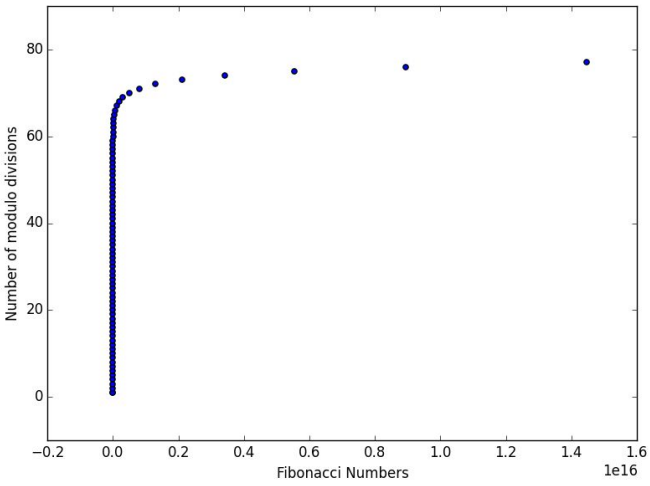
4660046610375530309 + 2880067194370816120 = 7540113804746346429 = fibonacci(93)

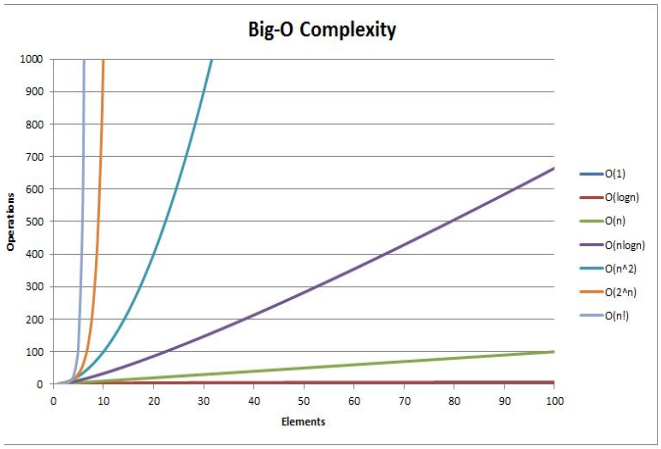
Where as fibonacci(94) = -6246583658587674878, which even if it is treated as a positive integer, can be proved invalid as:

6,246,583,658,587,674,878 < 7,540,113,804,746,346,429

Thus, when implementing Euclid's algorithm (gcd(m, x)) where m = F(k + 1) and n = F(k) we could set our upper bound of k at 92.

We measured the efficiency of the algorithm with the new input by counting the number of divisions taken to produce the greatest common divisor. This differs slightly from measurement through average case efficiency in that we didn't start the value of n (gcd(m, n)) at one and increment up. Instead our values of m and n were the current number of the Fibonacci Sequence and the number preceding it.

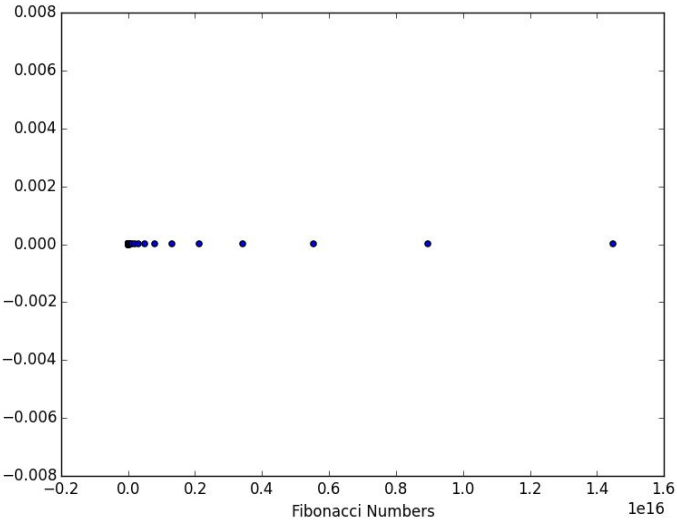


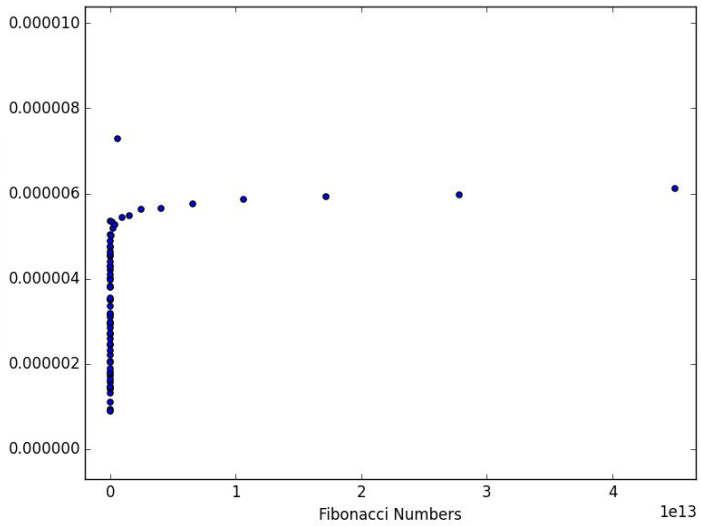


In the above diagram we can see a scatterplot of the number of modulo divisions required to compute Euclid's Algorithm with the Fibonacci Sequence as input. We see a dramatic increase in the number of modulo divisions as the input size increases. The curvature at the top of the scatterplot can be attributed to the fact that the numbers of the Fibonacci Sequence increase in size by an exponential factor. We believe this function to be in O(n!) complexity as it increases dramatically without even a slight curvature at the bottom of the "line".

This varies greatly from the efficiency of Euclid's Algorithm with standard input ($\Theta(logn)$), in that if the input didn't increase so dramatically we could expect to see a vertical asymptote in the graph of the Euclid's Algorithm with the Fibonacci Sequence. This would indicate that the number of modulo divisions possible would not have a limit, where as their appears to be a limit in the scatterplot with the standard input.

If we measure time instead of the number of modulo divisions we can produce a scatterplot that looks like this (zoomed in version on the right) :

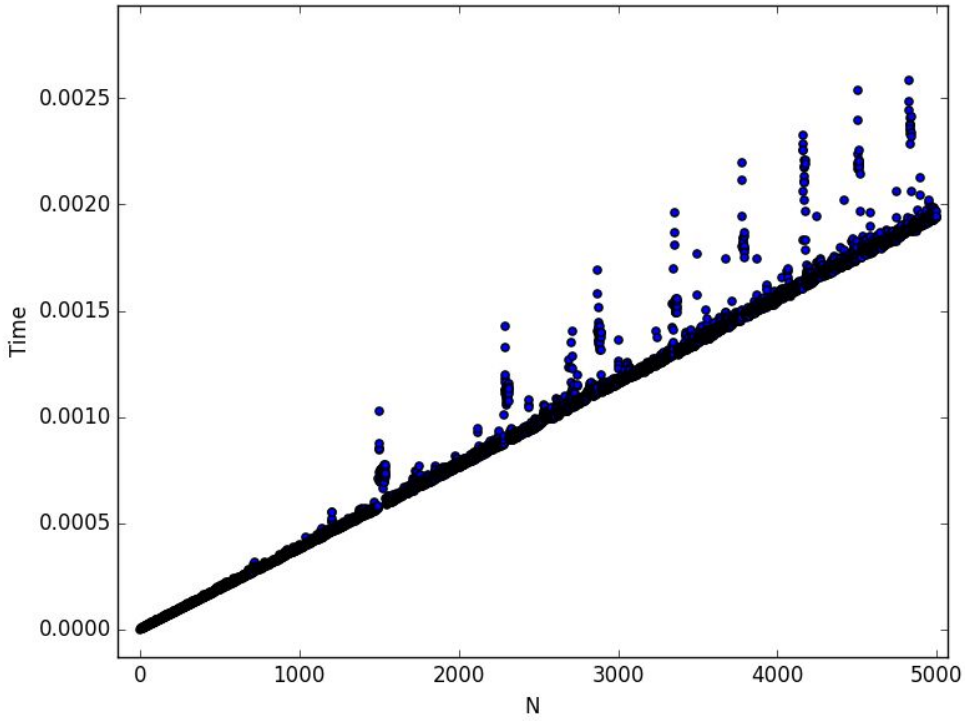




In the zoomed in image of our scatterplot we can see an image that seems to mirror our scatterplot with of the number of modulo divisions. However, there is one key difference, the scatterplot measuring time has outliers where as the scatterplot measuring modulo divisions had none. Thus, we can say that recording efficiency through modulo divisions is more accurate, but both will bring us to the same conclusion.

Task 3:

We chose to ascertain the efficiency of this algorithm by measuring runtime instead of counting the basic operation. We saw in the previous task that while there are some outlier data points, we can still reach the same conclusion as to which efficiency class this algorithm falls in.



This graph may appear linear as the majority of the values fall along a linear line. However, the values that lie above the line get further from the main line as the input size increases. Thus, if we plotted an average line, it would have a curvature as the input size increases. Therefore, we can deduce that the line falls into $\Theta(n \cdot f(n))$, where f(n) represents another term. We made the assumption that n > f(n) >= logn, as it is similar to the line of nlogn but has slightly greater curvature.

The algorithm we have designed is within $\Theta(n)$ complexity. It iterates through the two lists comparing the current indexes to each other. We increase the pointer to the next index of the value found to be lower in value. We then compare the indexes again. If they're equal we will add that value to another list of all common values. We continue this until one list ends. At this point we can be certain that there are no more common values. In the worst case, the max

number of basic operations is no larger than N where N is equal to the size of list A + list B. For an input of A = [1, 2, 3, 4, 5, 10] and B = [5, 6, 7, 8, 9, 10], we will get a C of [5, 10] and it will perform the basic operation 10 times on an input size of 12. The design of our common algorithm limits the traversal of our lists to a maximum of once per list. With an input of two lists of size M where list A is a list of consecutive even numbers and list B is a list of consecutive odd numbers, the worst case, is still within $O(n)$. The loop count of such an input is N - 1 (2M - 1).