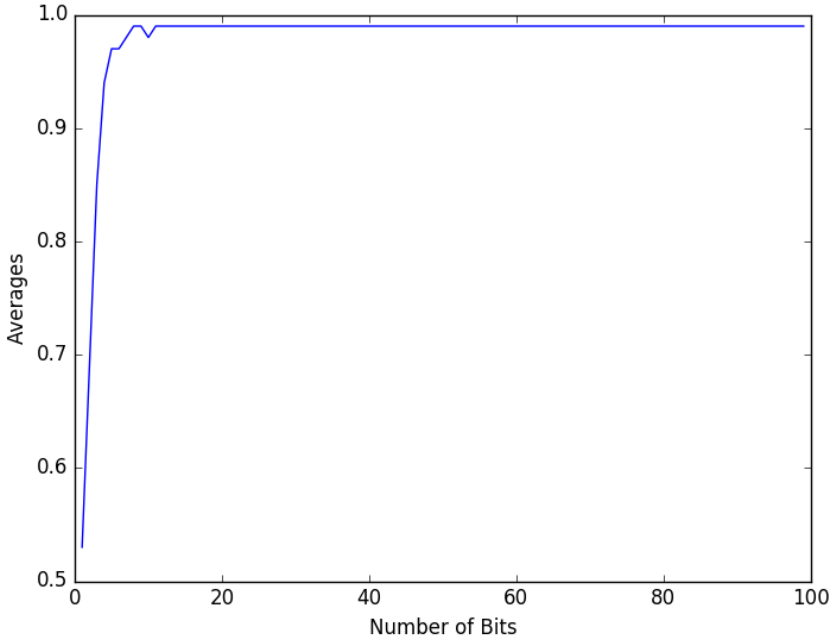
Joshua Steward

Lowell Batacan

Analysis of Euclid's Extended Algorithm Time Complexity

Euclid's algorithm is a classic example of an easy to understand algorithm, but constant reminder in almost any software system of just how important algorithms are in today's world. This observation can easily be seen in how the GCD is needed in cryptography and almost any developed system. That being said, Euclid's extended algorithm takes it a step further, and calculates an x and a y such that the original inputs, denoted a and b, can be plugged in such that ax + by = gcd(a, b). While slightly more mathematically advanced, this extended algorithm makes a good test to make headway into understanding how efficiency in terms of data size and efficiency in terms of iterative runs correlate.

Euclid's algorithm works by taking two inputs, a and b, and calculating the GCD of these two numbers. It does this iteratively by setting the value of one value to the value of the previous, the next value to the value of the previous value, then taking the remainder of the two new numbers. When this remainder hits 0, it means a GCD has been found, and the algorithm terminates. The extended version of this algorithm essentially does the same thing, but it adds functionality to this algorithm. It keeps the quotient of the previously calculates two numbers for each iteration, and uses that quotient to find the coefficients of Bezout's Identity. Then, when of the two newly assigned values hits 0, it terminates, and the coefficients are found as well. So, because the calculated coefficients in each run do not affect the calculated remainder, we can then conclude that the extended algorithm should have roughly the same run time and efficiency as Euclid's algorithm, with some extra memory assignments per iteration of course.



In this experiment, we have implemented the extended algorithm, and added in code that will keep track of the number of times the algorithm will run based on given inputs. We allow the user to enter a bound that will be the maximum number of bits used in any given number that is put into the algorithm. Then, random numbers are generated within this bounded number of bits. The algorithm for our experiment runs as follows: for all n in the number of bits the user specifies, the list that will be the bits axis in our plot has the current number of bits used appended to it. Inside this loop, for 100 runs of the algorithm, we will calculate 2 numbers that are bounded by the current number of bits. Then, the those runs spat out by the algorithm will be added to the average, which will be divided by the 100 runs after all 100 runs to get the total average number of runs. This average is then appended to the list of averages that will be used as our y-axis in our plot. This runs until every amount of bits until the maximum number of bits specified by the user is reached. Using this data, a graph of the number of bits by the iterations is generated.

This generated graph shows us several things. First, the number of runs goes up as the bounded bits for the randomly generated numbers goes up. This is to be expected. In fact, the number of runs increased very rapidly initially. However, what was unexpected was that this graph runs logarithmically. This means that as the number of bits gets higher and higher, the average number of runs drops off faster and faster. This phenomena showcases that Euclid's extended algorithm has an extremely high efficiency; the time efficiency is a logarithmic function, for O(log n). In short, the operations per run

decrease with each run. This makes sense, as the function cuts the inputs down by at least the b input's size every iteration. This makes running the algorithm on high memory usage objects very efficient.

In conclusion, this experiment has proven that not only does Euclid's algorithm have an O(log n) time efficiency, but that Euclid's extended algorithm has this same time complexity. We have shown that it terminates, and that in each iteration, it cuts down the potential size very efficiently. This makes this algorithm very efficient in large scale systems.