CS 415: Algorithms Analysis Spring 2019

Report by Jevan, and Juan | Section 1

Problem Statement

Task 1: Compute and compare the average-case efficiency of *Euclid's algorithm* and *Consecutive integer checking algorithm* (based on the above definitions). Produce a *scatterplot* of $MD_{avg}(n)$ and $D_{avg}(n)$ and indicate each algorithm's likely average-case efficiency class. Note that $MD_{avg}(n)$ and $D_{avg}(n)$ are functions of n; your program must choose the different values of n to generate the scatterplot.

Task 2: Compute the worst-case efficiency of *Euclid's algorithm* to compute GCD. Note that the worst-case inputs for Euclid's algorithm happen to be the consecutive elements of the Fibonacci sequence. Therefore, to implement this task, first generate the Fibonacci sequence using the iterative algorithm. Then, compute GCD(m, n) where m = F(k+1) and n = F(k) for several values of k >= 1 (Think about what should be the upper bound of k and clearly indicate it in your report)?

- Produce a *scatterplot* showing the number of modulo divisions taken to compute GCD as function of **m** and indicate the algorithm's likely worst-case efficiency class.
- How does this compare to average-case efficiency of *Euclid's algorithm* computed in Task 1?
- How does your analysis change if instead of measuring number of modulo divisions, you measure the actual time taken by the program to output the result?

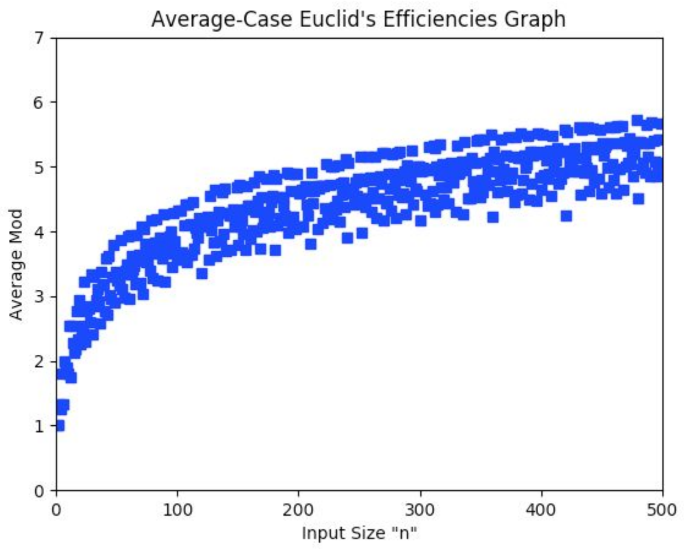
Task 3: The "Middle-school procedure" (Section 1.1) for computing the GCD requires i) Computing prime factors and ii) Determining common prime factors.

- The algorithm to find the common factors stored in two lists A and B should run in Θ(g) time where g = maximum of the size of A and B.
- Demonstrate its Θ(g) complexity by running your algorithm for several different values of of m and n and showing linear complexity in a scatterplot.

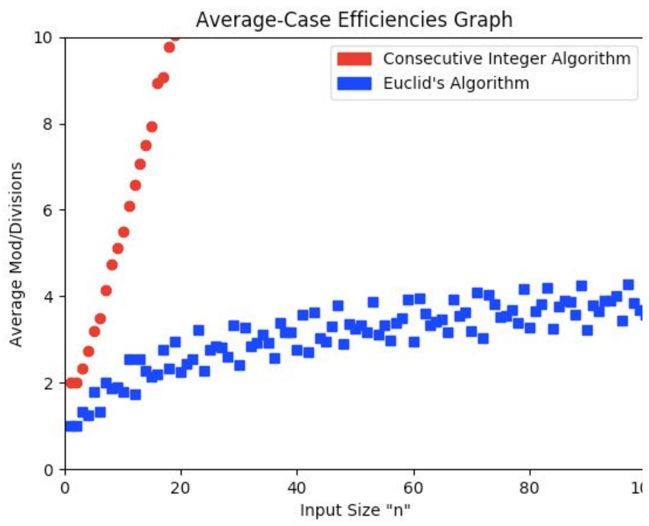
Problem Analysis

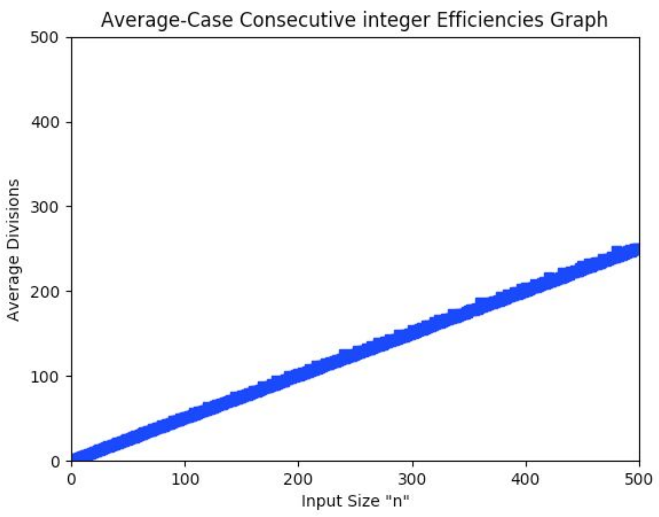
Task 1:

The average-case efficiency of *Euclid's algorithm* is defined by “the average number of modulo divisions MD$_{avg}$(n) made by the algorithm in computing gcd(n, 1), gcd(n, 2), . . . , gcd(n, n)”, and results in having an average-case efficiency class of Θ(log n) based on the curve of its scatter plot.



The average-case efficiency of the "Consecutive integer checking algorithm" is defined by “the average number of modulo divisions MD$_{avg}$(n) made by the algorithm in computing gcd(n, 1), gcd(n, 2), . . . , gcd(n, n)”, and results in having an average-case efficiency of Θ(n).





Task 1 (Continued):

Euclid's algorithm is a decrease and conquer algorithm, specifically of variable-size decrease type. Since gcd(m,n) = gcd(n, m mod n), at each step, m mod n will decrease the value to at least less than half of m and then you repeat the process until n = 0. Thus achieving Θ(log n) complexity.

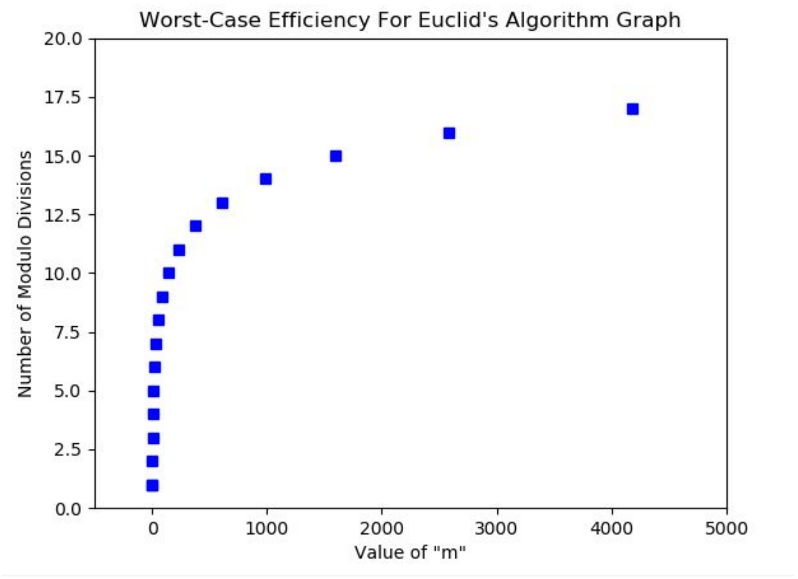
The consecutive integer checking algorithm just takes the min{m,n}, and see if it divides one of them first with remainder 0, and if it does, it checks the other number for a remainder of 0. If the first division fails, it subtracts 1 from the min{m,n} and repeats the the process until both achieve a remainder of 0. For the value of n, the average number of divisions was ~ n/2 ∈ Θ(n).

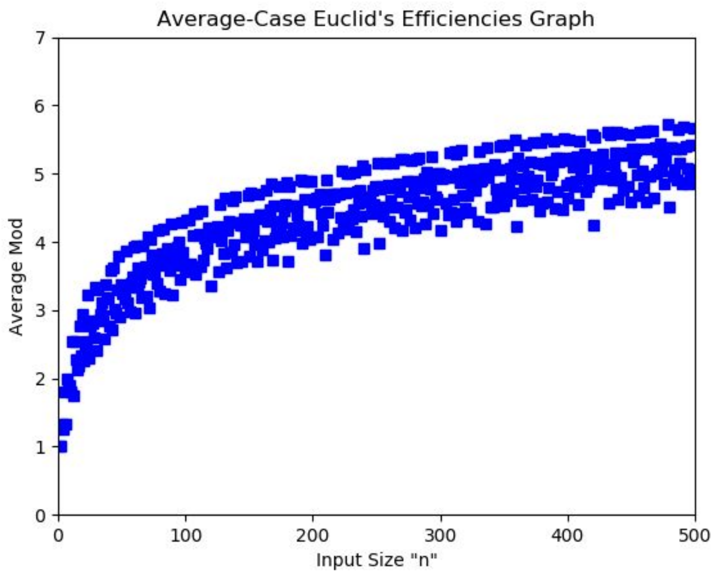
Since in the consecutive integer checking algorithm, we are reducing the size of the problem by 1, it will take much longer compared to *Euclid's algorithm* since in *Euclid's* we cut the problem by at least less than half of the input. In the consecutive integer checking algorithm, there are cases where we could have checked numbers from min{m,n} to 1 meaning we had to check m or n numbers. In each checking, there is at least 1 to at most 2 divisions, so min{m,n} to 2*min{m,n} divisions.

For the shared scatterplot above, we used values of n (x-axis), 1 to 200, so $MD_{avg}(1)$ up to $MD_{avg}(200)$ and $D_{avg}(1)$ up to $D_{avg}(200)$ (y-axis), but it only displays up to n = 100. For each separate scatterplot, we used values of n, 1 to 500 and display all of it.

Task 2:

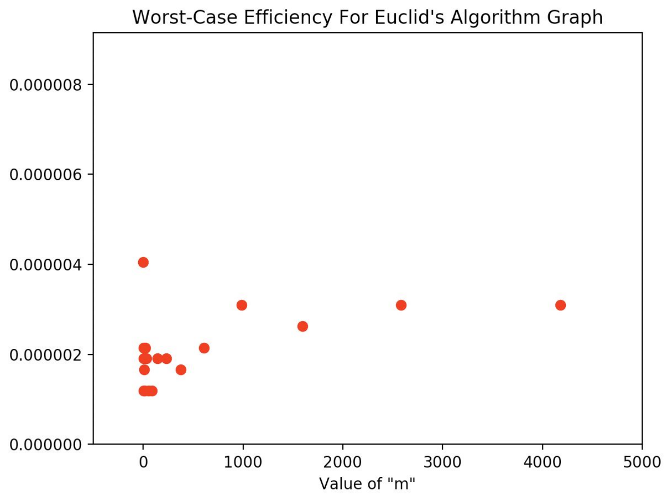
For computing worst case efficiency of Euclid's algorithm we use the fibonacci sequence, which is a combination of numbers 0, 1, 1, 2, 3, 5…, n-1. These numbers are found by adding the current number x_i by the next number x_{i+1} to get a fibonacci number, for example $x_{i+2} = x_i + x_{i+1}$. Using this logic we designed a function to generate a fibonacci sequence up to a given value k. After setting up our fibonacci function we then incorporated the sequence into our GCD(m, n) function where *m = fib(k+1)* and *n = fib(k).* Interestly, after running many tests we found k = 92 to be our upper bound, primarily because once you go beyond 92 the fibonacci number returned goes beyond the scope of a 64 bit number. Above is a graph showing the time complexity of Euclid's algorithm in the worst case. This graph seems to suggest the worst case for Euclid's algorithm using the fibonacci sequence is *O*(log n). When comparing Euclid worst case graph with the average case, we found a striking similarity in shape., leading us to conclude that its average case also has a *O*(log n) complexity.

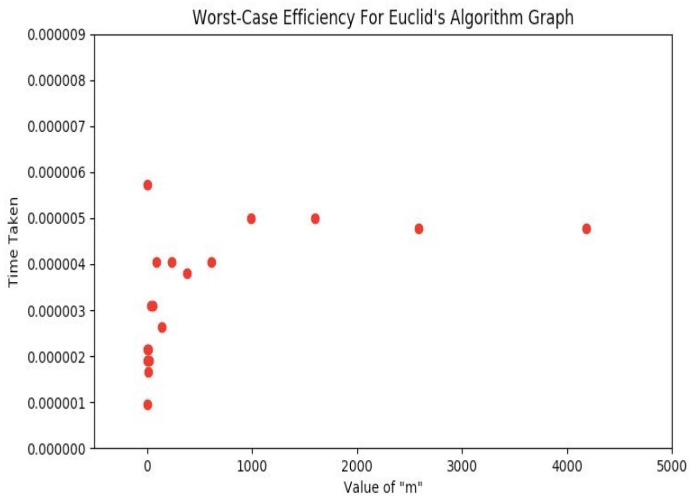




Task 2 (Continued):

Below we have a few graphs measuring the actual time taken by our program to output results for Euclid’s worst case. In both graphs the x axis uses values of m, remember values of m are essentially m = fib(k+1). To our surprise they appear to show a similar curve to our worst case Euclid’s algorithm above. We had initially thought measuring actual time taken would yield a linear increase over time, but surprisingly we were wrong. Though time in this case shows a similar curve to our worst case graph above, we cannot solely rely on time measurements simply because no machine performs the same. In the graphs below each result was recorded from a different computer illustrating an obvious difference in the outcome of the graphs. Though similarities do exist we cannot form a solid conclusion on the basis of these graphs below.





Task 3:

For computing GCD using the middle school procedure we started by finding prime numbers less than or equal to k. To find prime numbers we used Sieve's algorithm. After finding prime numbers we then had to compute prime factors for a given value. Using our Sieve's algorithm we generated a list of prime numbers to use in generating our prime factors for a given value x. Once we had our prime numbers and prime factors, we then built an algorithm to find common factors between two prime factor lists.The algorithm works by iterating the top value or bottom value based on a few comparison. Below is a step by step process showing how our common factors algorithm works, proving our algorithm runs in "Θ(g) time where g = maximum of the size of A and B".

Common Factors Algorithm Example:

Let's say we have 60's prime factors and 24's prime factors

60 = 2 * 2 * 3 * 5

24 = 2 * 2 * 2 * 3

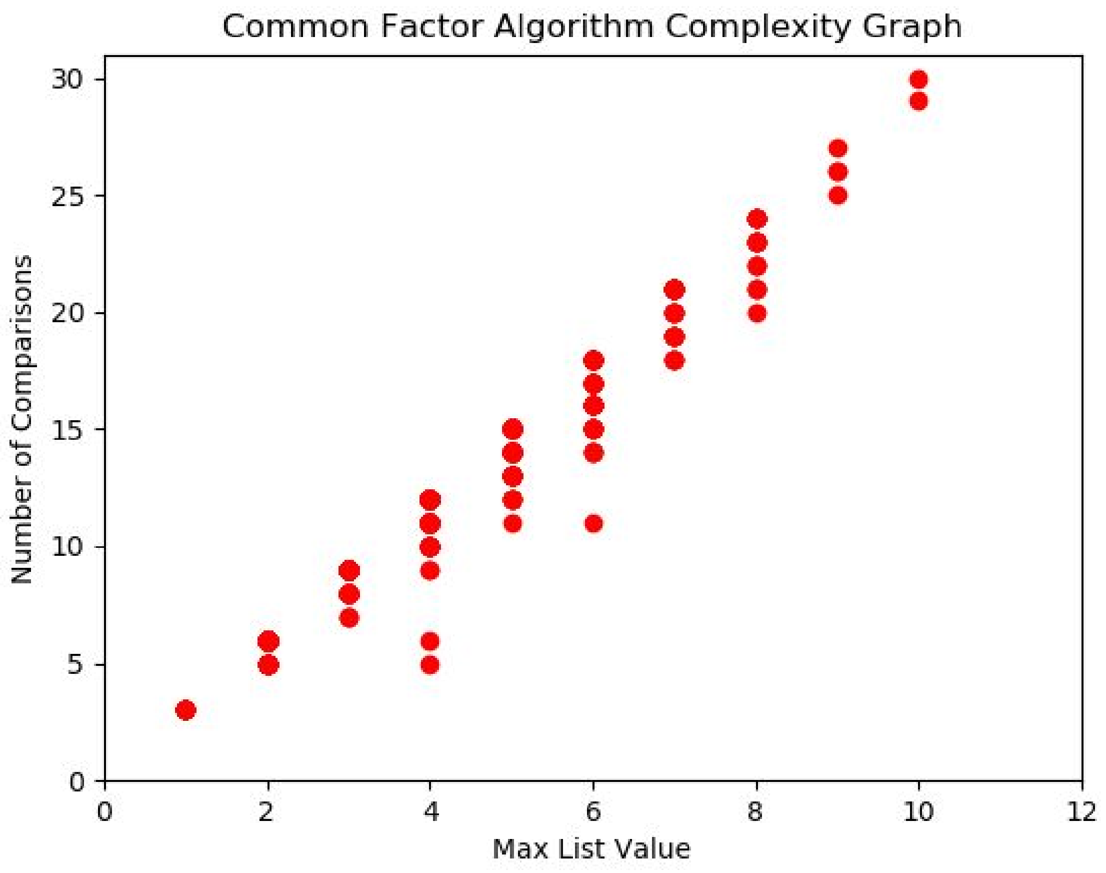
Color **blue** indicates index position in the list

2, 2, 3, 5 **2**, 2, 2, 3 Match: **Yes** 2 = 2 *Move both values to the right* Common factor List: **None**	**2**, 3, 5 **2**, 2, 3 Match: **Yes** 2 = 2 *Move both values to the right* Common factor List: **2**	**3**, 5 **2**, 3 Match: **No** 3 > 2 *Shift bottom value to the right* Common factor List: **2, 2**	**3**, 5 2, **3** Match: **Yes** 3 = 3 *Move both values to the right* Common factor List: **2, 2**	5 2 *End, because Iterator = size of the max list which is 4* Common factor List: **2, 2, 3**

* Note: this algorithm only took n steps.

Task 3 (Continued):

After finishing our common factors algorithm we started to work on a scatter plot to demonstrate that our algorithm satisfies Θ(g) complexity. To achieve this we built a scatter plot showing number of comparisons made in our common factors algorithm, and the max list size chosen between m and n prime factor lists. For the graph below our program selects numbers between 2 and 4000 for both values of m and n. Thus our graph represented here showcases our algorithm running linearly with a complexity of Θ(n).


Common Factor Algorithm Complexity Graph
Number of Comparisons
Max List Value