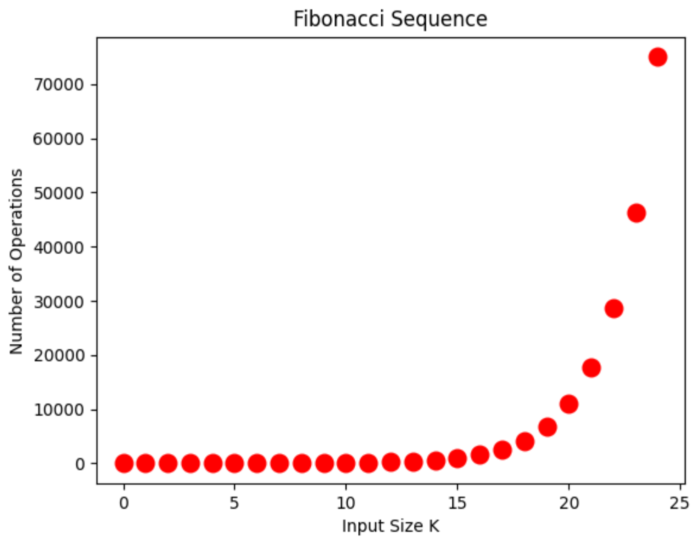
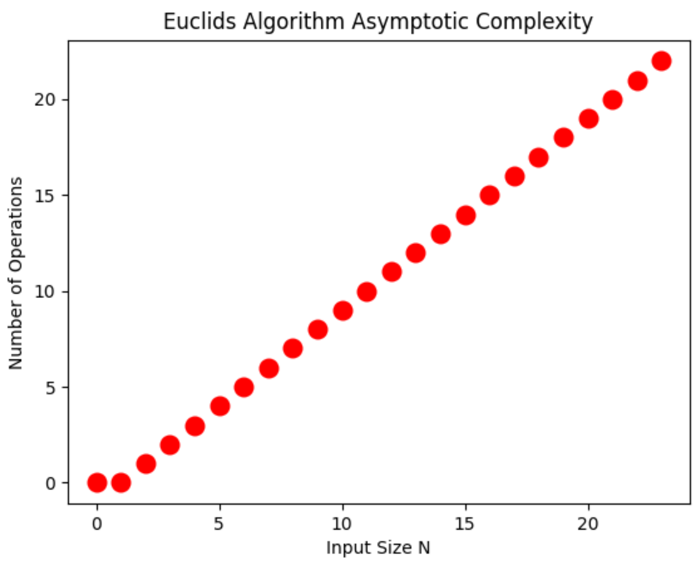
Alejandro Madrigal
Owen Mastropietro
CS 415
Spring 22

Project01a Report

While the user mode required less work, the scatterplot mode seemed to really push the concepts of the growth rates and helped visualize the differences in algorithms that perform the same tasks in a different manner. In task 2 we were able to compare different methods for the same solution that yielded different growth rates and allowed for a large difference in the upper bound for our input value. In task 3 we were again able to analyze different growth rates for the same solution that had a different approach. It still provided a different view though in that, first of all it was iterative, it illustrated the concepts of different cases.
**Note: we decided to plot our input size according to n as opposed to the more accurate binary representation of n = 2^b.

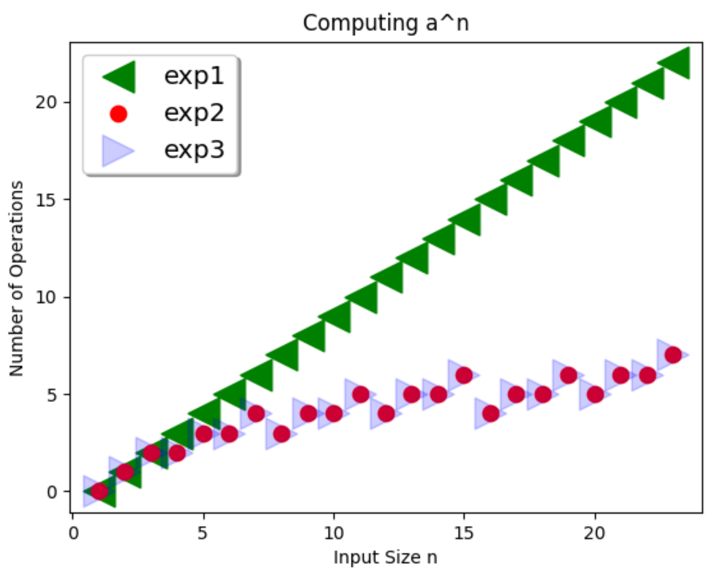
Task 1:
Fib = Θ(phi^n) or Θ(phi^2^b)
Gcd = Θ(log n) or Θ(b) but Θ(n) or Θ(2^b) in worst case (such as consecutive fib numbers)





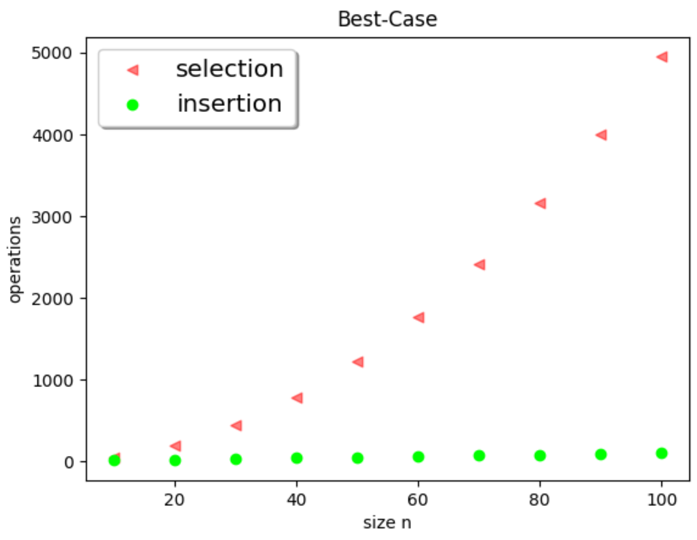
The first task was split into two subproblems that helped build our understanding of two recursive algorithms and their growth rates. One of these was to implement a recursive algorithm for computing the kth number of Fibonacci's famous sequence. Understanding the growth rate for this algorithm was difficult when we first came across it, so implementing it and getting to see not only its number of operations for given input sizes but how it affected Euclid's method for computing the gcd of two numbers helped visualize the homogeneous second-order linear recurrences with constant coefficients that we were building in class. Note: calculus 3 offered here at SSU has an interesting section that dives into the other methods of solving these (as described in Appendix B) without the trick the book uses. The largest value for k on task 1 that we were able to use was 35. We had to wait a few seconds before seeing output and anything beyond that became "unreasonable" for our scope. Given any integer, k, and computing the gcd of two consecutive fibonacci numbers based on that k value, we would be using two numbers that are already expensive to acquire from our exponential algorithm for computing the kth fibonacci sequence, and then, using those numbers, we would be hitting the worst case for euclidGCD which is Θ(n) (where n = 2^b) which is extremely costly. We chose, similarly for the rest of our algorithms/plots, to use the decimal representation of n (in this case k just to match the project instructions/formatting) for the scatter plot as opposed to the more accurate binary representation of the input size, n.

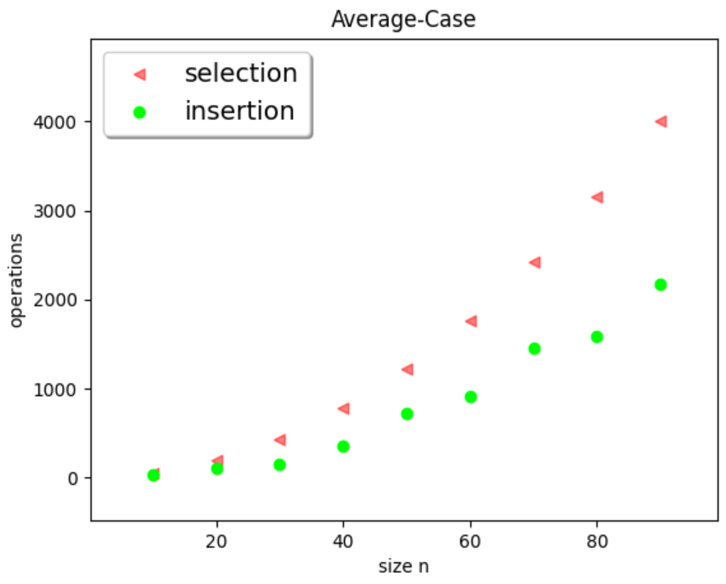
Task 2:
Decrease by one: Θ(n) or Θ(2^b)
Decrease by constant factor: Θ(log n) or Θ(b)
Divide and conquer: Θ(n log n) or Θ(2^b * b)

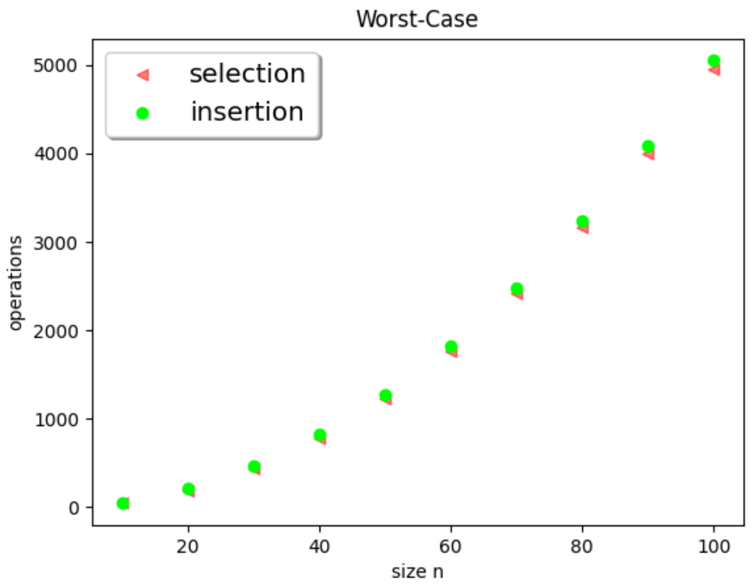


This task was very helpful in clearing up any discrepancies between decreasing by a constant factor and divide and conquer. The most immediate outcome would be just that. We were able to analyze how two seemingly identical methods were actually very different when looking into the recursive calls/trees of each algorithm. Despite mathematically being the same (effectively squaring both values), divide and conquer, with its quickly growing recurrence tree, illustrated how many more operations are being performed in the same amount of time as the decrease by a constant factor version. This means that, according to our graphs and the way we analyze efficiency, the decrease and conquer method is much more efficient with its lack of additional recursive steps. Furthermore, and somewhat surprisingly, the divide and conquer algorithm would eventually become less efficient than even the decrease by one method. Another learning outcome from this task was being able to apply the general approach and master's theorem to these algorithms, helping to solidify our understanding of analyzing different algorithms with different tools. A difficulty we had is that a lot of sources online seem to have different understandings and definitions for divide and conquer vs decrease by a constant factor regardless of the algorithm at hand - similar to the multiple different views of upper/lower bounds vs best/worst cases that are found on the internet.

Task 3:
Selection sort: best = worst = average = Θ(n^2) or Θ(2^b^2)
Insertion sort: best = Θ(n) or Θ(2^b), worst = average = Θ(n^2)


Best-Case
selection
insertion
operations
size n
5000
4000
3000
2000
1000
0
20
40
60
80
100


Average-Case
selection
insertion
operations
size n
4000
3000
2000
1000
0
20
40
60
80



While we didn’t really have any problems with choosing input values that were too large for exponentiation, we did experience similar difficulties as in task 1 when we were looking at the data/testSet files because of the amount of files and the poor growth rates of insertion and selection sort. We were able to test the graphs and the outputs for the number of operations per a given input value with our summation formulas for the growth rates to confirm that selection sort provided Θ(n^2), again not using the binary representation, in each of the cases while insertion sort at least had the off chance of providing a more efficient best case of Θ(n) and even with the average case being slightly better (but only be a constant factor.)