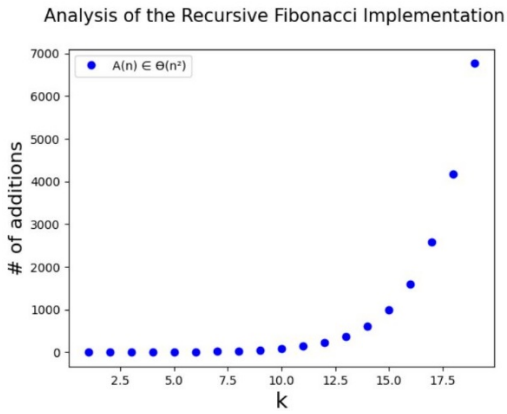
Feedback Given From: Guy

Task 1: Fibonacci Sequence and GCD



make graph a bit bigger

Analysis for Recursive Fibonacci

Since there is only one input of a given 'size', in this algorithm its natural input size is the "magnitude" of n, which is the number of bits, $\log_2 n$.

TYPO ↑ O

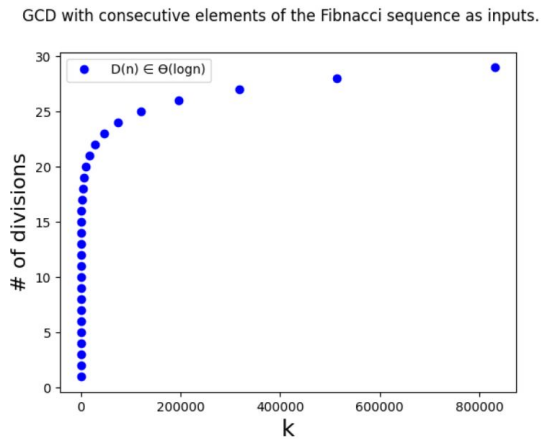
The basic operation in this algorithm is the addition because this is the operation that gets executed the most, and $A(n) \in \Theta(n^2)$

The recursive implementation ~~of the~~ Fibonacci asymptotic complexity ~~is~~ gives an of $\Theta(n^2)$ because $A(n) \in \Theta(n^2)$

For the task, I chose a value of 30 as my n because the running time did not fare well for large values. A benefit of having a recursive implementation such as this one, is that it works well for smaller values which is why I am working with that particular size. When you need a quick solution, this is a simple recipe for a program, but as n gets larger the running time is quadratic. When trying to populate my array to plot the GCD worst case algorithm, I wanted to get a bigger input size to get enough data to represent that algorithm. Using the iterative solution was an option, but I instead used the formula provided by the book and the one discussed in class.

The running time doesn't become quadratic as n grows bigger, it is quadratic for this algorithm.

GCD WORST CASE



The inputs natural size for this equation is the n. The basic operation in this case would be the modulo division, as it is the operation that happens most often.

Was the number itself.

At first I was considering my input size to be the values that were used

We've discussed when the input size is just an integer, than

D(n) ∈ Θ (logn)

The time complexity in the worst case for Euclid's Algorithm is log(n).

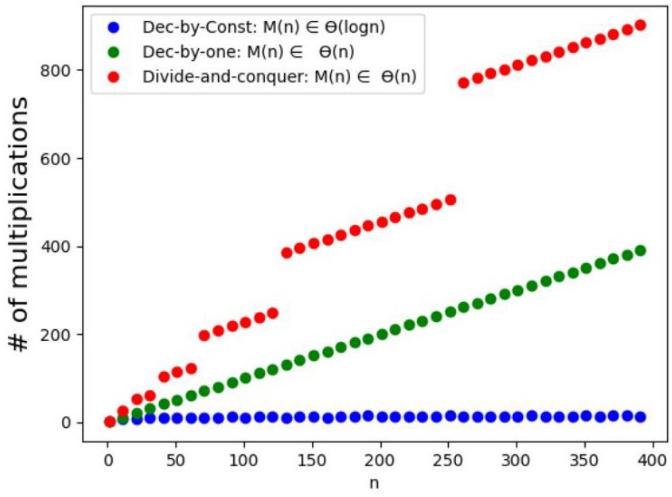
I originally did my graph with the my input as log(n) and so I got a linear curve. The reasoning behind my understanding comes from the recurrence relation, and its recursive tree. I know from solving the recurrence relation numerous times, we know that the number of basic operations is (n-2), and when drawing the recursive tree for the algorithm, coupled with how we solved decrease by constant, the number of basic operations is related to $Ø^n$. So, to compute number of divisions for $Φ^n$, it is n. If what we were graphing was the input to the GCD algorithm, we know that that's log(n) that is the magnitude of bits, for an algorithm given an integer. And in that case, we would get linear curve. I initially graph my input as $Φ^n$, which game me a linear curve. $M = Φ^n$. After taking the log of both sides, you get log(m) = nlog (Φ), since Phi is Φ = 1.618, the golden ration you can remove it, so **log(m) = n**

So, my assumption is that n, in this case is my input, so therefore M(n) ∈ Θ (log m) but if I were to assume m was my input it would give me a linear curve.

A notable observation I had was in user testing mode when it would print out the number of modulo divisions to calculate GCD (k+1, k), given any value of It would always give back 1. I know that generally for inputs other than the inputs from the Fibonacci sequence, it will not need to always go to recursively go down and call the function recursively for all k. The fact that each result is 1, this means winds down the whole entire recursive call. And that's the recurrence relation comes to mind, 1+ gcd(f(n-1),fib(n)).

Perhaps combine these two statements together?

Task 2:



Decrease by one is notably the most efficient one, only differing by a constant with Divide conquer. These two algorithms natural input size will be n, as this grows constant to the number of basic operations.

Dec by constant natural size on the other hand will be log(n) representing the number of bits.

Also clear shown, is the linear curve. While there are gaps in divide and conquer, I believe this just do because it is dividing int's input size, but either way all of plots are linear the complexity is n.

When taking a closer look and zooming in, it is clear the decrease by constant is log(n).

ADD GRAPH

Because the graphs are all incredibly efficient and none of the are quadratic, I decided to pick a large value for n. My value was 400, and this worked incredibly fast because these algorithms fare well as n gets larger.

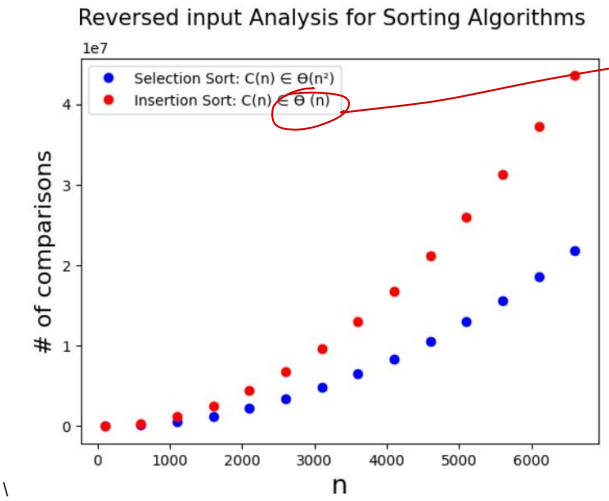
The basic operation for each of them is Multiplication. And the natural input size, because we are giving it one value will be log2(n), and getting more than one linearly.

Task 3:

Worse Case:

Move to next Page

With reversed input, Selection Sort and Insertion sorted have the same complexities which is $C(n) \in \Theta(n^2)$.



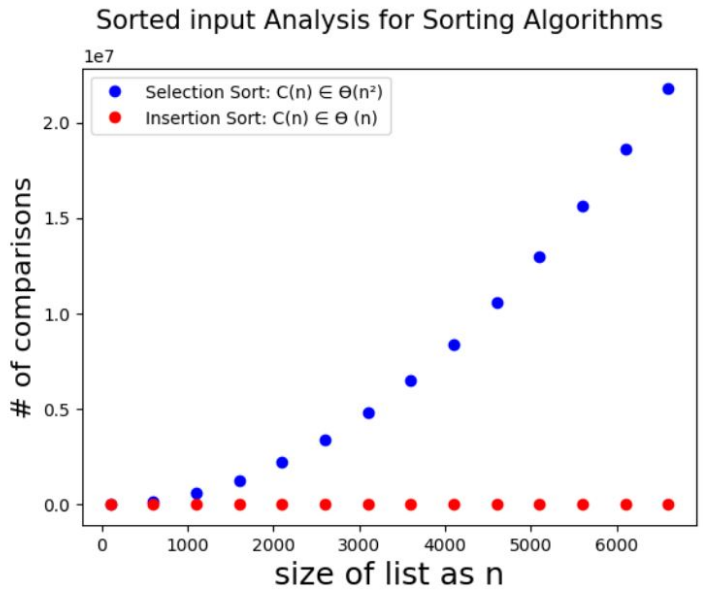
The natural input size for both algorithms is the magnitude in numbers of bits, $\log_2 n$Obvious by the graph both these algorithms grow exponentially. Selection sort seems to be a little slower than insertion sort, but that is just by a constant, so in the worst case we know the upper bound, and lower bound of the function, therefor we can claim that we understand this function running time which gives us $\Theta(n^2)$. In selection sort, the inner loop only does one swap, each pass.

In selection sort it does as many swaps as possible, until an element is in place. So there for selection sort, will have half as many comparisons.

Also, when reviewing these algorithms, it is clear the selection sort is not a stable algorithm. It will always switch the largest element with the last index, 1 less than the the currently sorted values, elements will be out of place.

Insertion sort on the other hand is stable, as they retain equivalence keys, as it doesn't swap equal keys, and the duplicate is always stored after one of them.

Best Case:



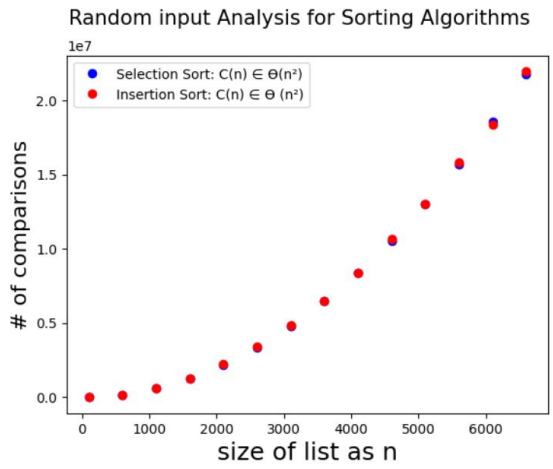
The natural size for insertion sort on the best case is n and for selection sort it is $\log_2 n$.

The basic operations for both are the number of comparisons.

We can see that insertion sort dominates this with Θ(n). The reasoning being, since everything is in order, it initially passes through each element in the array, does zero swaps then breaks out of the loop. For selection sort on the other hand, it will always swap the last element with the largest one depending on the implementation, and on each iteration compare the first one witch each element in the this that has yet to be sorted. So, for insertion sort on a sorted input, $C(n) \in \Theta(n^2)$.

Average Case:

For the random selection you can see that they look almost identical, as they have many overlapping points on the scatter plot. The running time for Selection Sort and Insertion sort for the average case is $C(n) \in \Theta(n^2)$.



Maybe mention how they're both n^2 as the scatterplots overlap.

My Selected Ranges for Computing are from 1000-6000 with an increment of 500.

With ranges from 100-10000, with an increment of 500, it took a significant amount of time to compute due to each algorithms complexity, other than Sorted Input for insertion sort which is Θ (n) all the other complexities for every other case is Θ (n^2). ← run on Sentence.SPLIT into two.

Having the quadratic functions to computer, meant as it got to 10000, it would take minutes just to get that initially solved. The only way that I could get it to pass over 10009, I had to increment it to 2000 every time, and by the time I got to 10000, which was a little bit of time by the way, it took like a solid minute to finish computing. With a range of 6000, I am still clearly able to observe the behavior.

This Paragraph is hard to read/follow