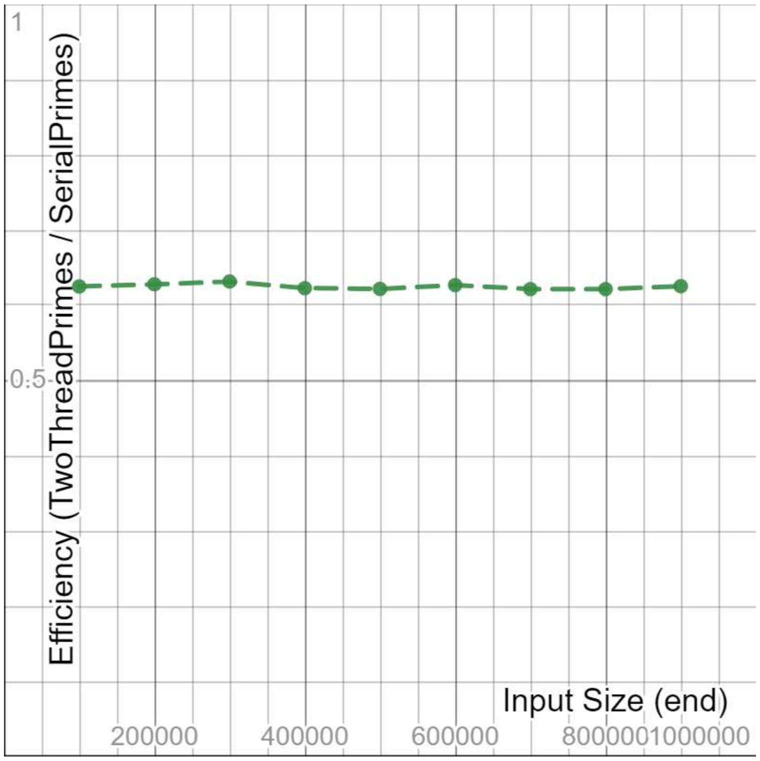
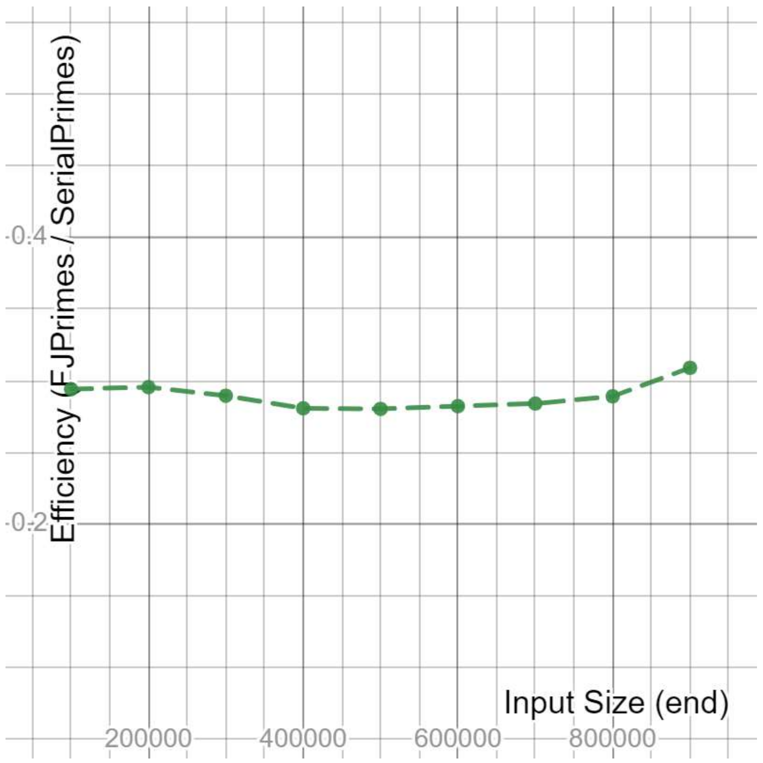
PrimesFJ Lab Report

My SerialPrimes algorithm works by, for each number between `start` and `end` (inclusive), checking if the number is a prime, increasing a counter if it is. It does so by checking if it evenly divides each number between 2 and its square root (inclusive), stopping early if it discovers that the number is a composite. I would assume that the order of growth for the algorithm would be O($n^{1.5}$), because the outer loop runs until n, while the inner loop runs until $\sqrt{n}$ (or O($n^{0.5}$), as I will refer to it for the rest of the write-up). When I tested it, my code had a doubling ratio of about 2.6, while my calculations suggest that an O($n^{1.5}$) algorithm should have a doubling ratio of about 2.8. It is probably due to random noise, but perhaps, since primes become less common at higher numbers, it is more likely that each number will be divisible by an earlier factor, allowing the algorithm to conclude without reaching $\sqrt{n}$.

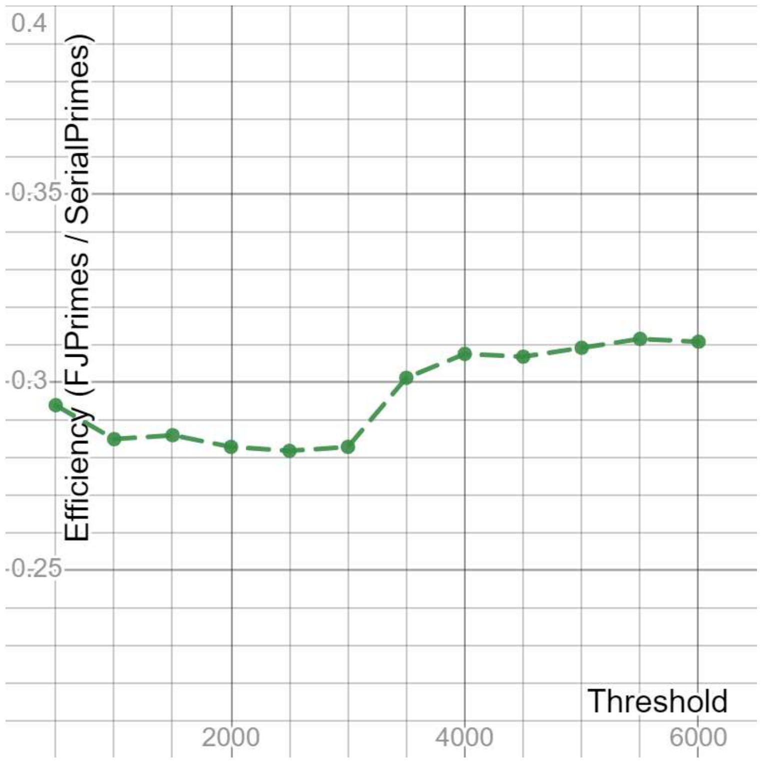
The theoretical best performance for TwoThreadPrimes is exactly half the runtime of SerialPrimes. This is because, by making use of two threads, my computer could use two cores instead of one to run the program, so it could run it up to twice as quickly. You can see my results for performance in the graph on the right. This isn't quite the maximum performance, which would be 0.5 (half the performance, a 200% increase), and is instead about 0.63, closer to a 160% increase. Its change in efficiency between different inputs is small and inconsistent, because doubling the number of cores used changes the efficiency by a constant factor, independent of input size. My calculated doubling ratio was 2.3, giving a performance somewhere between O(n) and O($n^{1.5}$).



My laptop has 2 cores, or 4 with hyperthreading. On a p-core machine, my PrimesFJ algorithm would have an order of growth of O($\frac{n^{1.5}}{p} + n^{0.5}$), where n is the size of the input. As can be seen on the graph on the right, on my computer, my algorithm has an approximately constant runtime ratio with SerialPrimes. The ratio varies a bit due to random noise, but averages at around 0.29, about a 345% improvement. This clearly shows that my virtual cores are being treated as processors. It is much less than the theoretical maximum of a 400% increase, likely due to parallelization inefficiencies such as the cost of thread creation.



To test the ideal threshold, I picked a sample `end` of 1,000,000 and tested various sizes. You can see the result on the right. There was not much variance, but if you zoom in (like I did on my graph), there is a noticeable change in efficiency at different thresholds. With a threshold of 3000 or less, efficiency is just over 0.28, or about 350%. If the threshold becomes 4000 or more, the runtime ratio suddenly rises to about 0.31, leaving efficiency at about 320%. That is why I used 2500 as my threshold. I'm not sure why the change in efficiency is so sudden, but the reason it exists is likely because lower thresholds have greater load balance. The observed order of growth was approximately O(*n*).



The three orders of growth do differ, with SerialPrimes being about O($n^{1.5}$), PrimesFJ being about O(n), and TwoThreadPrimes being somewhere in the middle. The reason for this is likely that since the theoretical order of growth with infinite cores would be O($n^{0.5}$), as more cores are being used to solve the problem, the order of growth becomes asymptotically closer to the order of growth of infinite cores. I suspect that when you test PrimesFJ with your 6-core computer, you will get a more efficient result than I did. But if the order of growth of these algorithms really was lower than that of SerialPrimes, the ratio between their runtimes and SerialPrimes' runtime should have decreased at higher values of n, as SerialPrimes becomes less efficient, which was not what I observed.