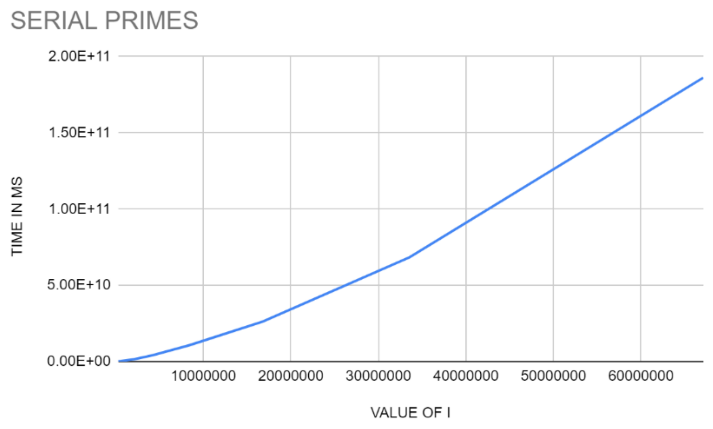
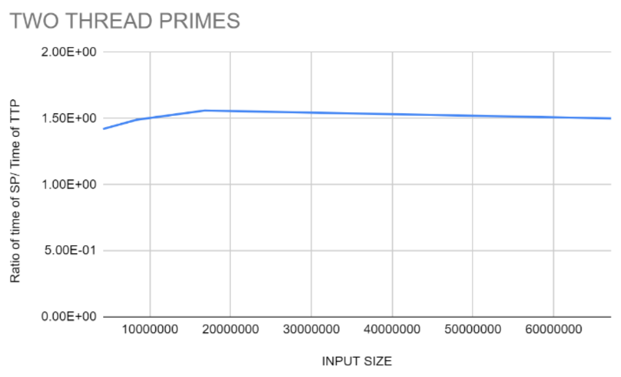
SerialPrimes:

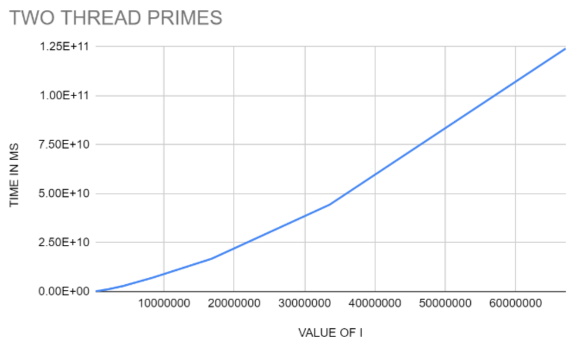
My code contains an outer loop than runs through each number from 2 to n, and an inner loop that runs from 3 to the square root of number that the outer loop is up to. I'm not a math guy, so I'm not sure exactly how to express this in an equation other than that my loop would run sqrt(2)+sqrt(3)+…+sqrt(n-1)+sqrt(n) times, or the summation of the square root of k from 3 to n. The closest upper bound I can think of for the order of growth would be n*sqrt(n), or a ratio of something less than 2.82. My actual average ratio was 2.59, which was around what was expected, given that this is an upper bound that is not big theta (see the graph below).



TwoThreadPrimes:

The theoretical best performance should be 2 times as fast as SerialPrimes. The actual ratio of time of SerialPrimes/ time of TwoThreadPrimes is 1.5 (see the graph below on the left). Perhaps the discrepancy is a result of the cost of creating those threads, so there is only a 150% increase in efficiency instead of 200%. The observed ratio of growth was an average of 2.41 (see the chart below on the right), which makes sense because cutting the time in half doesn't make a big-O improvement.

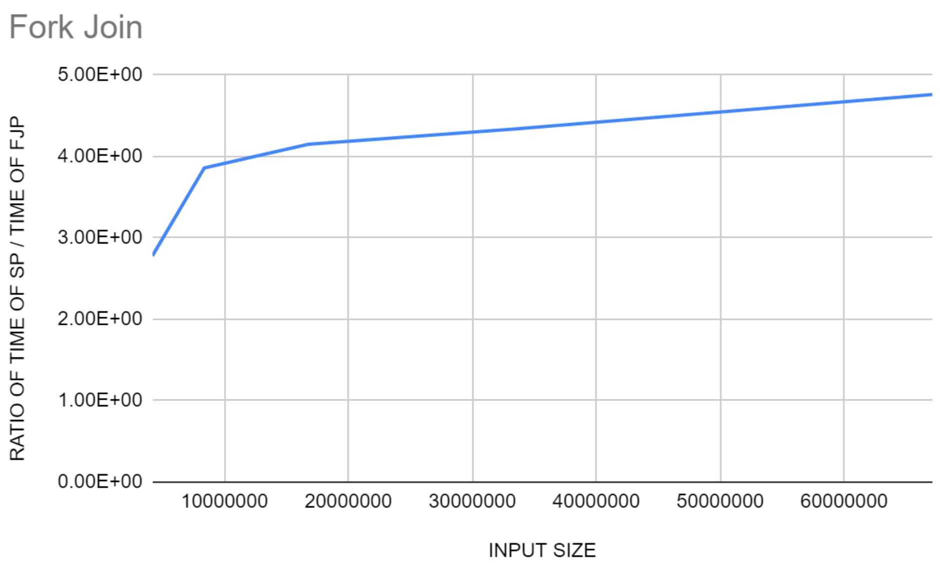




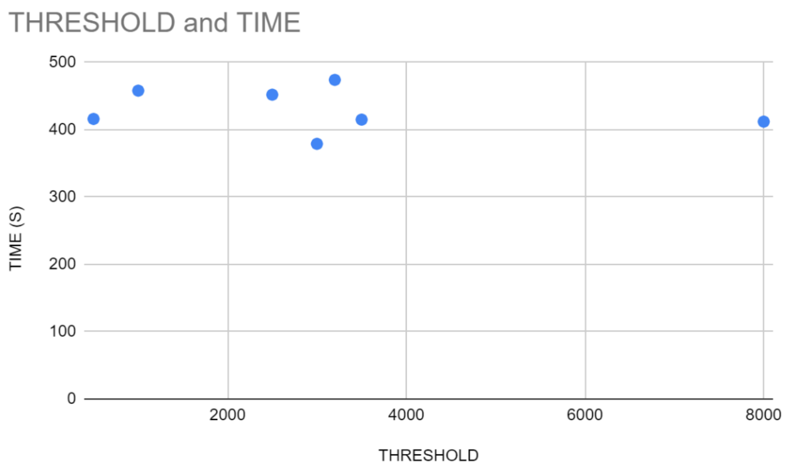
PrimesFJ:

My laptop has 4 cores, and 8 threads, but those 8 are really the Intel logical threads which makes ideal performance hard to predict. The theoretical best performance on an n-core machine is an (n*100)% increase in efficiency. When the start is held constant, and the end value is doubled, my ratio of time of

SP/ time of FJP was an average of 4 (see the chart below). This is a little hard to analyze, because I don't fully understand what an Intel logical thread is. It seems however, that any program I execute runs on one core and uses two "intel logical threads," so in a strange way, it actually might make sense that I experienced an efficiency increase of 400% on a 4 core, 8 logical processor machine. The order of growth should be O(logn+sqrt(n)), the amount of time it would take to partition until the threshold, plus the largest amount of sequential work that must be done, which is the sqrt(n) inner loop being performed on the highest value. As n gets to very large, the sqrt(n) will become the much more significant value, so we would expect a doubling ratio of something like sqrt(2) or 1.41. When tested though, my average doubling ratio was 1.9, not 1.41, so I'm not sure what exactly happened. Maybe it is a result of the overhead of the java fork-join framework.



When I tried a start value of 2 and an end of 500,000,000, the best threshold was 3000.



In terms of the order of growth of all of these algorithms, SerialPrimes is O(n*sqrt(n)) as is TwoThreadPrimes, because it just divides that in half. FJPrimes would knock that big-O to O(logn+sqrt(n)) because it takes logn to partition and then the largest amount of sequential work being done by a thread is sqrt(n).