Wren Paris-Moe
9/19/2019
CS 317
Professor Jeffery Miller

Lab Project 2 – Perfect Numbers

Introduction:

Studied in number theory, a perfect number has the property that it is equal to the sum of all its proper divisors. For example, the number 6 has proper divisors 1, 2, and 3 so that 1 + 2 + 3 = 6 meaning it is in fact a perfect number. Currently only 51 known perfect numbers exist with the most recent being found in 2018 with 49,724,095 digits. Due to the enormous size of the larger perfect numbers, they become increasingly difficult to produce. However, additional properties have been discovered that allowed for more perfect numbers to be found. This lab takes an in depth look at those properties and additional methods that may increase the amount of perfect numbers your processor is able to compute.

The first method examined in this lab attempts to compute as many perfect numbers as possible using the basic algorithm which tests if the sum of proper divisors is equal to a given number *n*. The letter *n* signifies any number from the outmost for loop used as the basis for computing perfect numbers in each algorithm. The second method used a known property proven by the Euclid-Euler theorem that relates perfect numbers to Mersenne primes (M_p) which have the form $2^p - 1$. The letter *p* is used to denote any *n* that is found to be prime. The algorithm works as follows: if *n* is a given prime *p*, and if $2^p - 1$ is prime, then $(2^p - 1)(2^{p-1})$ is a perfect number. A basic primality test by trial division determined if n or $2^n - 1$ were prime. Method 2 was named MersenneProperty in Java and Python. The next algorithm attempted to improve upon the last by utilizing the Lucas-Lehmer primality test for Mersenne numbers. Using the Lucas-Lehmer test, let *p* be an odd prime. Then M_p is prime if and only if $S_{n-1} = 0 \pmod{M_p}$ where $S_1 = 4$ and $S_n = (S_{n-1})^2 - 2$. The last three algorithms were implemented on top of the Lucas-Lehmer approach to see if improvements could be made. The first of these was multi-threading. Multi-threading allows tasks to be divided among several threads to be run simultaneously. For select programs this can greatly speed up runtimes while for others it will make little to no difference. Later it is examined whether multi-threading is applicable to computing perfect numbers efficiently. For the purposes of this lab, different ranges of *n* were sent to various threads to check for perfect numbers concurrently. This way a single thread does not need to iterate through the range of all *n*. Several thread counts were tested to determine the efficiency of this approach. The next method deleted redundancy from the program. A list of prime numbers was computed before the main algorithm was run so that every *n* is not tested for basic primality. The last improvement tested if manipulating numbers at the bit level would speed up the process. This takes advantage of the fact that for a given prime *p*, the Mersenne number $2^p - 1$ takes the form 111…111 in binary (a sequence of *p* 1's) while the multiplier, 2^{p-1}, takes the form 100…000 (a 1 followed by *p* – 1 0's). All approaches were written in Python and Java to compare the success of improvements in separate programming languages. (Note: Times are not comparable cross-language since Python's time.perf_counter_ns() function and Java's System.nanoTime() method calculate relative time in different ways)

Hypothesis:

It was hypothesized that each ensuing algorithm for generating perfect numbers would improve upon the last. The first method would be the least effective since it is forced to iterate through (n / 2) + 1 terms to find all proper divisors for all n. The second method would improve on this since it is required to check less terms in order to see if n and $2^n - 1$ are prime. For example, if n is even, the prime checking method is broken immediately after dividing the number by 2. There was uncertainty around whether the third method using the Lucas-Lehmer test would improve the last approach since there was confusion about how the primality test worked. It was assumed that each of the next three improvements would improve upon the Lucas-Lehmer approach. There was uncertainty that the improvements in Python would be as effective as in Java since the language is further removed from the processor level.

Methods/Results (Java):

First a look at the programs in Java. Due to the immense size of perfect numbers, the BigInteger class was used to compute them. This class was used in all algorithms, including those which did not exceed the int or long range. This way runtimes would be comparable. Each program was broken once ten minutes had passed since the last perfect number was found. Each algorithm was run twice and an average of the two data sets was used to achieve accurate results. Later the Lucas-Lehmer approach was run for eight hours to see how many total perfect numbers could be found.

Basic Algorithm & Method Two:

The basic algorithm and second method were the least effective, so they were examined separately from other approaches. After the first edition of method two was written, two improvements were made and implemented in the succeeding algorithms. The effectiveness of each version was tested by comparing the runtimes to find eight perfect numbers. The original version of method two used a basic primality test that iterated from i = 2 to (n / 2) testing if n was divisible by each i. If it was found that n % i = 0, the test would be broken and true would be returned telling that n was not prime. The problem with this approach is that if n is prime, all (n / 2) terms must be iterated. An enhancement was made by changing the upper bound of the for loop to the square root of n + 1. This takes advantage of the fact that all proper divisors of n come in pairs, one on each side of the square root of n. Therefore, only one of the two divisors are needed to determine if n is prime. The last improvement was implemented on top of the last by testing if M_p was prime only if p was prime. This reduced the amount of times that the primality test was run making the method more effective. Data indicating the effects of each improvement on method two is given by Table 1.

Table 1. Elapsed time to find eight perfect numbers for three variations of method two in Python and Java. Time measured in nanoseconds and not comparable cross-language.

	Time to Find 8 Perfect Numbers (ns)	
Version of Method 2	**Python**	**Java**
Original Primality Test	94294226300	63426652400
Improved Primality Test	4707900	6376400
Reducing M_p primality test	4091700	5957800

Table 2. Elapsed time to find most perfect numbers using the basic algorithm and method two in Java. Time measured in nanoseconds.

	Time to Find (ns)	
Perfect Number	**Basic Algorithm**	**Method Two**
1	964000	848500
2	1824300	1251950
3	40976600	1357550
4	1967002200	1479750
5		1685000
6		2523400
7		4123100
8		5957800
9		32716200
10		1.30665E+11

The basic algorithm produced a total of four perfect numbers. Method two, using the known relationship to Mersenne primes, found ten perfect numbers with drastically faster runtimes than the basic algorithm. Notice the difference in time to find the fourth perfect number between each approach. The basic algorithm took 1,965,522,450 nanoseconds longer than the second method. There was a large jump in the time to find between the ninth and tenth perfect numbers for method two.

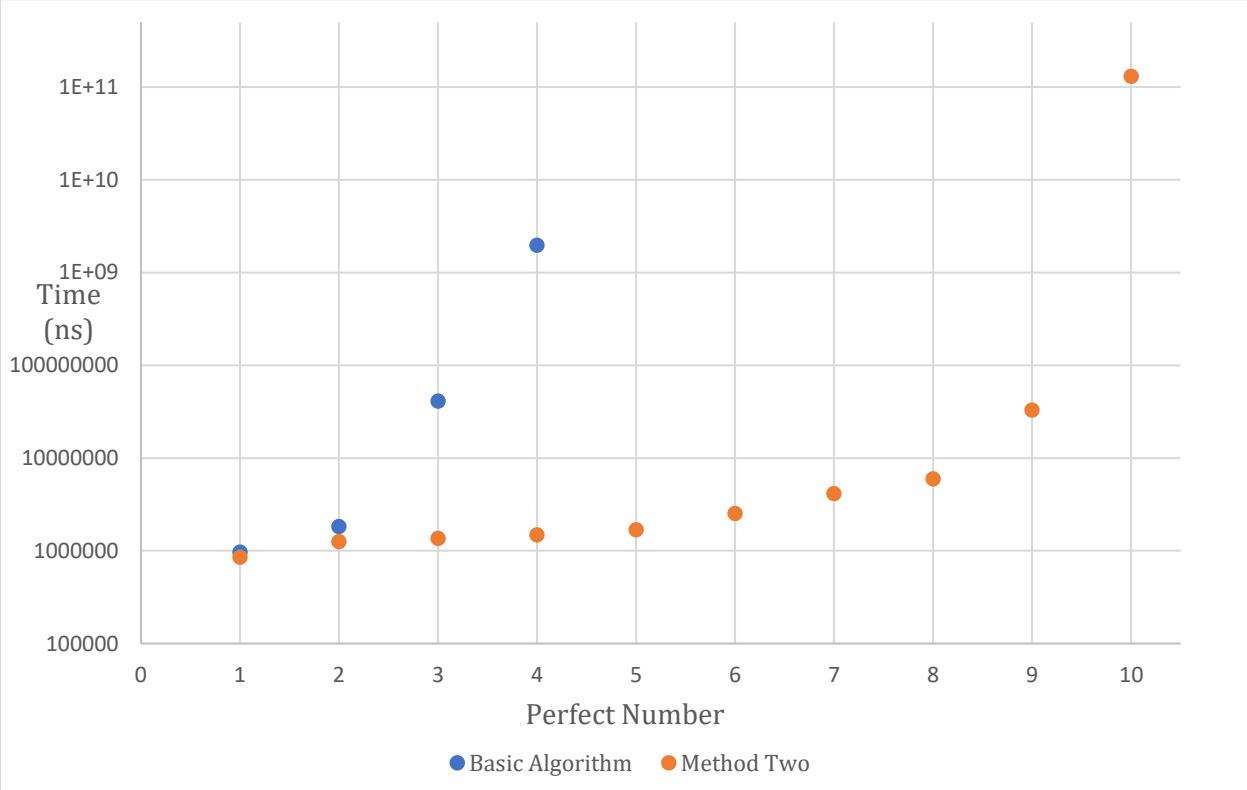

Figure 1. Elapsed time measured in nanoseconds to produce perfect numbers using the basic algorithm and method two in Java.

As shown in Figure 1, the times taken to find a given perfect number for the basic algorithm and method two. The horizontal axis reached ten perfect numbers though the basic algorithm found a maximum of four. The vertical axis has a logarithmic scale of base ten to better illustrate the disparity of times between algorithms. The data points for the basic algorithm appear to have a linear rate of increase. However, due to the logarithmic scale, this instead represents an accelerating rate of increase. An exponential trendline given by excel of $y = 29358e^{2.5975x}$ best fits the data. A general trend is less apparent in the data for method two. Neither an exponential nor polynomial equation could express a trend in the data with much accuracy.

Table 3. Time per digit measured in nanoseconds for each perfect number found by the basic algorithm and method two in Java.

		Time Per Digit (ns)	
Perfect Number	**Digits**	**Basic Algorithm**	**Method Two**
1	1	964000	848500
2	2	912150	625975
3	3	13658867	452517
4	4	491750550	369938
5	8		210625
6	10		252340
7	12		343592
8	19		313568
9	37		884222
10	54		2419723013

Table 3 holds the time per digit for each perfect number computed by the basic algorithm and method two. Table 3 was derived from Table 2 by using the addition of a “Digits” column. To create Table 3, the original times of Table 2 were divided by the digits in the corresponding row of Table 3 where the given perfect number was matched. This resulted in a new metric, the average nanoseconds taken to produce a digit for a given perfect number. As seen in Figure 2, plotting the times per digit allows for discrepancies in the original data to become more apparent.

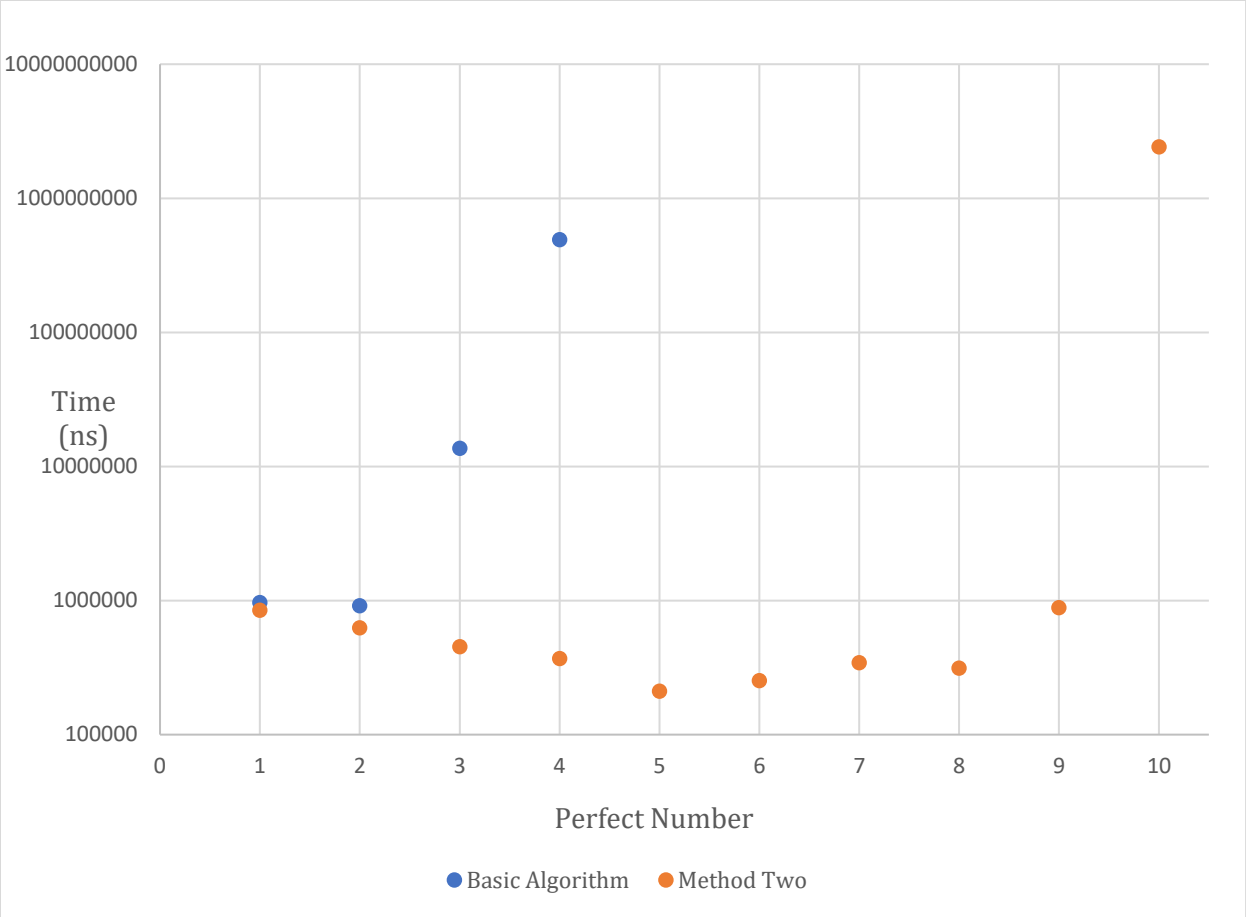

Figure 2. Time per digit to find each perfect number for the basic algorithm and method two in Java.

The data for time per digit fluctuated more than the regular times to find a given perfect number. This graph uses a logarithmic scale on the vertical axis with base ten. For the basic algorithm, the second point decreased from the first to then maintain a similar exponential trend to the original times. A polynomial trendline best fits the data with the equation $y = 1E+08x^2 - 4E+08x + 4E+08$ given by excel. The times per digit for the second algorithm are inconsistent compared to the method's regular times. A decreasing trend existed moving from the first perfect number to the fifth. The time per digit then increased for two consecutive numbers, decreased from to the seventh to the eighth, and finally increased at an accelerating rate for the last two points. A polynomial trendline is the closest match to method two but remains far from an accurate representation of the data's progression.

The two algorithms share an imbedded for loop format where the inner loop contains an if statement confirming if the number *n* from the outer loop is divisible by any integer *i* (*n* % *i*). The major difference is the purpose and implementation in each method. The basic algorithm uses the statement to find all proper divisors of *n*. That means for all *n,* a mod calculation occurs (n / 2) + 1 times. For method two, the statement is used as a primality test on *n*. Therefore, the for loop is broken the first time the statement is found to be true and moves on to the next *n*. Even though method two uses the primality test twice, it computes the mod function significantly less per any range of perfect numbers. If *n* is even, the for loop breaks as quickly as the second iteration. This

allows method two to be far more effective than the basic algorithm. The second method experiences an enormous jump in runtime from the ninth perfect number to the tenth. In fact, finding the tenth number took almost 4000 times as long the ninth. This illustrates where method two reaches its limit and that using the basic primality test for Mersenne numbers is no longer viable.

Lucas-Lehmer Approach:

The Lucas-Lehmer approach (referenced as LL) made the largest leap from the preceding algorithm. While method two found ten perfect numbers, the LL method produced a total of 26 when ran for eight hours. However, for purposes of this lab, the program was broken once ten minutes had passed since the last perfect number was found. This gave a total of 23 perfect numbers. Below is the Java code used to implement the Lucas-Lehmer primality test.

```
public static boolean LucasLehmer (int p, BigInteger Mp) {
        BigInteger s = BigInteger.valueOf(4);
        for (int i = 0; i < p - 2; i++) {
                s = s.pow(2).subtract(BigInteger.TWO).mod(Mp);
        }
        return s.compareTo(BigInteger.ZERO) == 0;
}
```

Table 4. Elapsed time for the Lucas-Lehmer method in Java to find 23 perfect numbers with the corresponding time per digit in nanoseconds.

Perfect Number	Digits	Time to Find (ns)	Time Per Digit (ns)
1	1	41100	41100
2	2	1109050	554525
3	3	1200600	400200
4	4	1280450	320113
5	8	1436600	179575
6	10	1549100	154910
7	12	1666950	138913
8	19	2132250	112224
9	37	6864900	185538
10	54	11765050	217871
11	65	13441600	206794
12	77	14754700	191619
13	314	72525350	230972
14	366	94657800	258628
15	770	482903850	627148
16	1327	2376125550	1790600
17	1373	2692964450	1961373
18	1937	7779356550	4016188
19	2561	19606232500	7655694
20	2663	22178660600	8328449
21	5834	292522719800	50141022
22	5985	322596417900	53900822
23	6751	470036833250	69624772

Table 4 provides the times to find each perfect number with the calculated times per digit for the LL method. There is a large jump in the number of digits going from the 12th to the 13th perfect number growing from 77 to 314 by a difference of 237. Demonstrating the magnitude of the numbers in this lab, the 22nd perfect number was comprised of almost 6,000 digits. Take note that the first perfect number, 6, is produced by n = 2 but the LL test follows the requirement $n \geq 3$. Therefore, the first perfect number was found independently from the rest. This is true for all methods implemented on top of the LL approach. Therefore, data for first perfect number was excluded from the figures for the remaining methods. The corresponding graphs for the LL approach are on the following page.

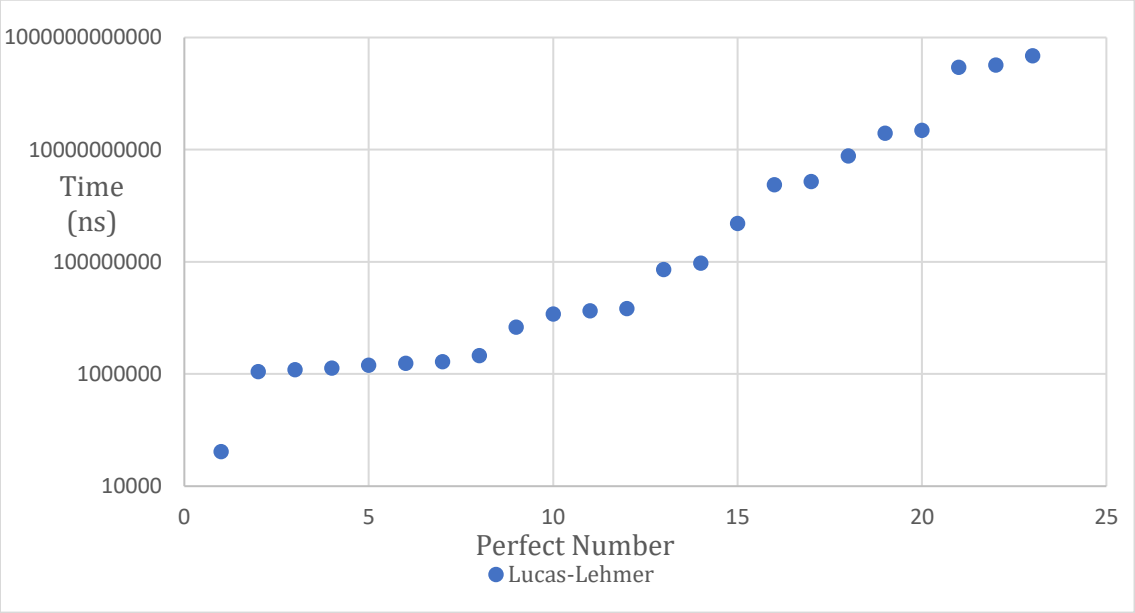

Figure 3. Elapsed time in nanoseconds to find each perfect number for the Lucas-Lehmer method.

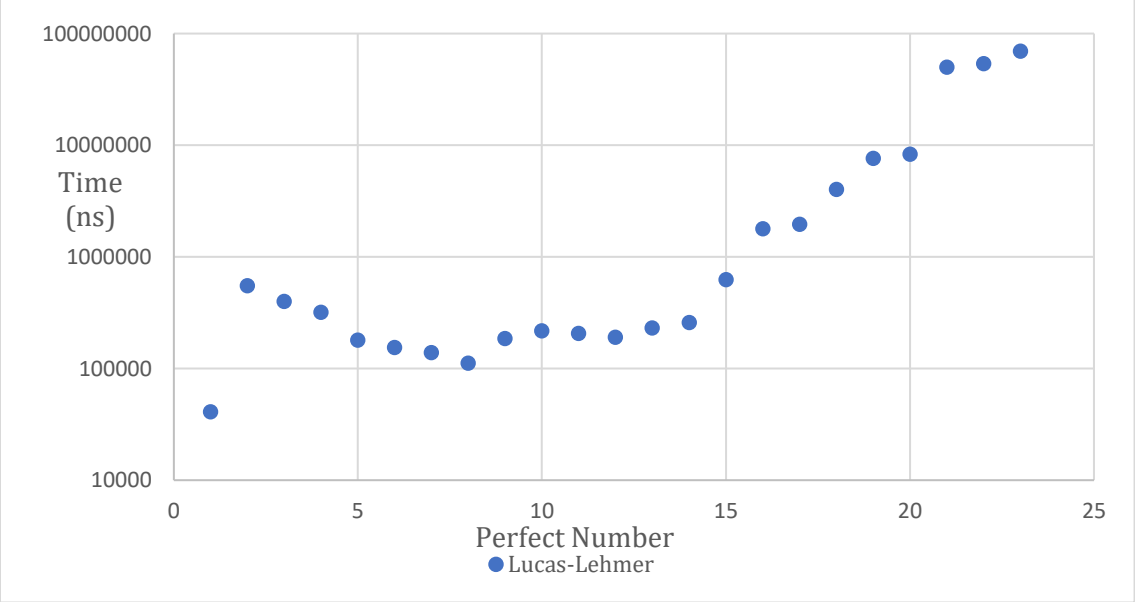

Figure 4. Time per digit in nanoseconds to find perfect numbers with the Lucas-Lehmer method

A logarithmic scale of base ten was used in Figure 3 and Figure 4 to create a better picture of the disparities between data points. Both graphs make it clear that the first perfect number is an outlier to the remaining data. Similar to method two, the data for time per digit fluctuated inconsistently. Excluding the first, the time per digit decreased from the second perfect number to the eighth. From that point on, an increasing trend existed with minor variance for the time per digit of each perfect number found. The best fit trendline given by excel for the time per digit was the polynomial equation $y = 290645x^2 - 5E+06x + 2E+07$. This illustrates the noticeable variation in the data. For the first graph, an exponential trendline of $y = 27354e^{0.6874x}$ fits the data showing an accelerating rate of increase for the time to find perfect numbers. Observe that a shared trend exists between Figure 3 and Figure 4 from the 13th perfect number onward.

The LL method made the largest difference of all the proposed improvements. In Java, the LL method was able to find 13 more perfect numbers than the second approach and 19 more than the basic algorithm. The only difference between the second and third methods was the use of the Lucas-Lehmer primality test for Mersenne primes. As p increases, M_p grows exponentially. Therefore, the basic primality test becomes increasingly costly for M_p's. The LL algorithm found the tenth perfect number over 10,000 times faster than method two. The LL test is faster than the regular primality test for M_p's because the for loop is iterated through far less times meaning less modulus functions are called. For only the tenth perfect number, the M_p given by $p = 89$ causes the basic for loop to iterate 24,879,108,095,804 times while the LL test has a total of 88 iterations. That means computing over 24 trillion modulus functions in method two solely when $n = 89$.

Multi-Threading:

Multi-threading was analyzed to determine if using multiple processors at once could decrease runtimes from the last approach. The LL method produced 23 perfect numbers before ten minutes passed without another being found. Therefore, each threading example was capped at 23 perfect numbers to allow for an easy comparison. Several thread count ranges were implemented to determine the true effectiveness of the multi-threading technique. For simplicity, the three groups of threads were called perfThread1, perfThread2, and perfThread3.

perfThread1 used a single thread to produce the first 23 perfect numbers within the given range of n: 2 to 70,000. Only one thread is utilized so really this is not an example of multi-threading. Instead this is used as a comparison to see if a threading the original LL approach will slow down runtimes. Therefore, this is nearly identical to running the LL method.

```
perfectThread thread1 = new perfectThread(2, 70000);
```

The second version utilized eight threads to split up the workload. This was the first step in determining whether an improvement could be made on past approaches. The ranges of n passed to each thread for perfThread2 were as follows:

```
perfectThread thread1 = new perfectThread(2, 1000);
perfectThread thread2 = new perfectThread(1000, 3500);
perfectThread thread3 = new perfectThread(3500, 5000);
perfectThread thread4 = new perfectThread(5000, 10000);
perfectThread thread5 = new perfectThread(10000, 11300);
perfectThread thread6 = new perfectThread(11300, 20000);
perfectThread thread7 = new perfectThread(20000, 22000);
```

perfectThread thread8 = new perfectThread(22000, 24000);

perfThread3 attempted to improve the runtimes of multi-threading by assigning a smaller number of threads to the following four ranges of *n*:

perfectThread thread1 = new perfectThread(2, 1000);
perfectThread thread2 = new perfectThread(1000, 4000);
perfectThread thread3 = new perfectThread(4000, 10000);
perfectThread thread4 = new perfectThread(10000, 25000);

Table 5. Elapsed time to find perfect numbers using the Lucas-Lehmer method and multi-threading in Java.

	Time to Find (ns)			
Perfect Number	**Lucas-Lehmer**	**perfThread1**	**perfThread2**	**perfThread3**
1	41100	81450	90300	81800
2	1109050	1259900	1981000	1432200
3	1200600	1347700	2263850	1705600
4	1280450	1410400	2352650	1787650
5	1436600	1540400	2520050	1930850
6	1549100	1645400	2692600	2084750
7	1666950	1764100	2894800	2265150
8	2132250	2201600	3511500	2835100
9	6864900	7147850	10282150	9120400
10	11765050	11707100	18262700	14641650
11	13441600	13292000	21203200	17579300
12	14754700	14296750	23326350	19728800
13	72525350	76304900	178716250	143895450
14	94657800	101585650	292403000	231375050
15	482903850	475028100	645329300	550903450
16	2376125550	2326173950	3899470850	3300699800
17	2692964450	2638358750	4339577300	3697181150
18	7779356550	7770400250	11803788400	8551825050
19	19606232500	19493360850	15000224700	11590951000
20	22178660600	22113199400	18288227000	13569984750
21	2.92523E+11	2.94398E+11	1.82634E+11	1.64317E+11
22	3.22596E+11	3.235E+11	3.01421E+11	2.82538E+11
23	4.70037E+11	4.69234E+11	3.33458E+11	3.12317E+11

Results from each version of multi-threading in Table 5 were compared to the LL approach to see if improvements could be made. perfThread1 was nearly identical to the LL method. A ratio relating perfThread1 and the LL approach was computed between all perfect numbers. The smallest was 0.88 while the largest ratio was 1.03. This proves how identical the two datasets are. The first perfect number was excluded from the ratios due to its outlying character.

perfTread2 took longer to produce eighteen perfect numbers than the LL approach. However, the runtimes for the last five perfect numbers and especially the 23rd were faster.

Though, due to the different thread ranges being run simultaneously, several perfect numbers were found out of place. The 19th and 20th perfect numbers were produced before the 18th while the 21st and 22nd were found before the 23rd.

The third and final arrangement of threads maintained a similar trend to perfThread2. perfThread3 improved runtimes for the last five perfect numbers but was slower for the first 18. Compared to the eight-threaded approach, the use of four threads resulted in faster runtimes for all 23 perfect numbers. Like its predecessor, perfThread3 failed to print all numbers in the correct order. The 23rd perfect number was produced prior to the 21st and 22nd due to the specific arrangement of thread ranges.

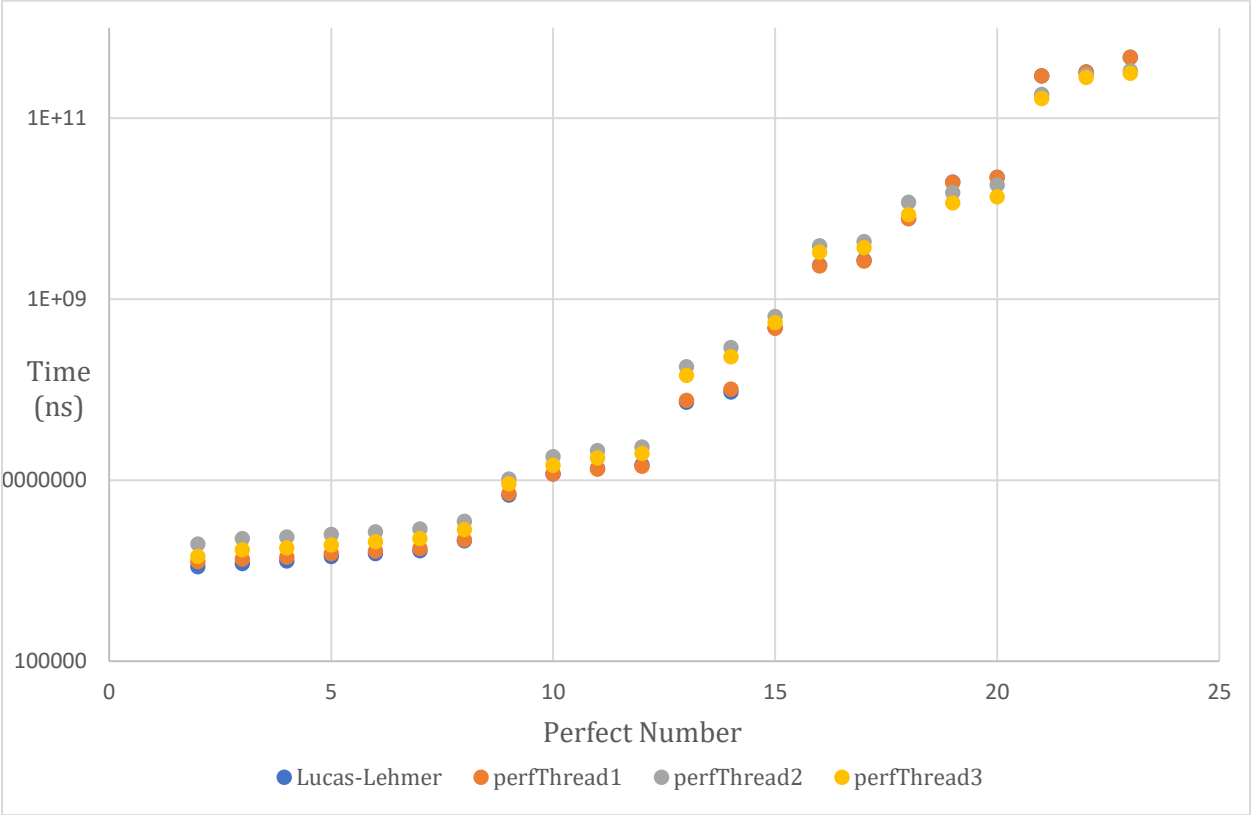

Figure 5. Elapsed time for the LL method, perfThread1, perfThread2, and perfThread3 in Java to find the first 23 perfect numbers.

Runtimes to produce 23 perfect numbers for the LL method and each multi-threaded approach are plotted in Figure 5. The first perfect number was excluded from the graph due to it being a major outlier. Points for the LL method are not visibly apparent due to its similarity to perfThread1. By examining the table, there seems to be a large difference in runtimes among the LL method and the last two versions of multi-threading. However, the scatter plot for these methods contradicts that observation. The graph illustrates that, in relation to the immense size of the numbers, the difference between runtimes is insignificant. In fact, there is an almost identical pattern within each of the datasets. Progressing to the next perfect number, each method seems to increase by an equivalent amount. This is especially true for the first 12 perfect numbers and is validated by inspecting the table. Further verifying this is the similarity of trendlines given by excel for each method:

Lucas-Lehmer: $y = 28994e^{0.6838x}$ --- $R^2 = 0.9376$

perfThread1: $y = 31904e^{0.6784x}$ --- $R^2 = 0.9348$

perfThread2: $y = 69884e^{0.6429x}$ --- $R^2 = 0.9441$

perfThread3: $y = 52297e^{0.6493x}$ --- $R^2 = 0.9458$

Shown by the R^2 values, an exponential trendline fits each dataset with high accuracy. Specifically, the exactness of exponents affirms the similarity of patterns within the data. However, the coefficients vary by a significant amount. Therefore, the time per digit for each perfect number is needed to further understand the multi-threading results.

Since multi-threading runs several ranges of *n* simultaneously, the order that the perfect numbers were produced was not always in sequence. This must be considered when calculating time per digit. Therefore, the method used on page four to match perfect numbers to their corresponding row and digits must be reevaluated. Now, the particular sequence of perfect numbers from perfThread2 and perfThread3 must be examined to determine the correct pairings between times and digits. Only once matching the correct values to each perfect number can the appropriate times per digit be calculated.

Table 6. Time per digit in nanoseconds for perfect numbers in proper order produced by the Lucas-Lehmer method and multi-threading in Java.

		Time Per Digit (ns)			
Perfect Number	**Digits**	**Lucas-Lehmer**	**perfThread1**	**perfThread2**	**perfThread3**
1	1	41100	81450	90300	81800
2	2	554525	629950	990500	716100
3	3	400200	449233	754617	568533
4	4	320113	352600	588163	446913
5	8	179575	192550	315006	241356
6	10	154910	164540	269260	208475
7	12	138913	147008	241233	188763
8	19	112224	115874	184816	149216
9	37	185538	193185	277896	246497
10	54	217871	216798	338198	271142
11	65	206794	204492	326203	270451
12	77	191619	185672	302940	256218
13	314	230972	243009	569160	458266
14	366	258628	277556	798915	632172
15	770	627148	616920	838090	715459
16	1327	1790600	1752957	2938561	2487340
17	1373	1961373	1921601	3160654	2692776
18	1937	4016188	4011564	9441521	4414985
19	2561	7655694	7611621	4609054	4525947
20	2663	8328449	8303868	5632829	5095751
21	5834	50141022	50462535	51666336	48429484
22	5985	53900822	54051732	55715595	52183287
23	6751	69624772	69505850	27052935	24339689

perfThread2 and perfThread3 found the 23rd number before the 21st and 22nd resulting in a surprisingly low time per digit shown in the last row of Table 6. Due to the variability in thread ranges, this exists for every perfect number computed out of order. For example, in Table 6, the times per digit comparing the 18th and 19th perfect numbers dropped from 9,441,521 to 4,609,054 nanoseconds. That value was less than half the time per digit of the previous perfect number. The LL method and perfThread1 were more efficient and had a lower time per digit for smaller perfect numbers. However, once the larger perfect numbers were reached, that trend was reversed causing perfThread2 and perfThread3 to become more efficient. For instance, in perfThread3, the time per digit for the 23rd number was three times smaller than in both the LL method and perfThread1.

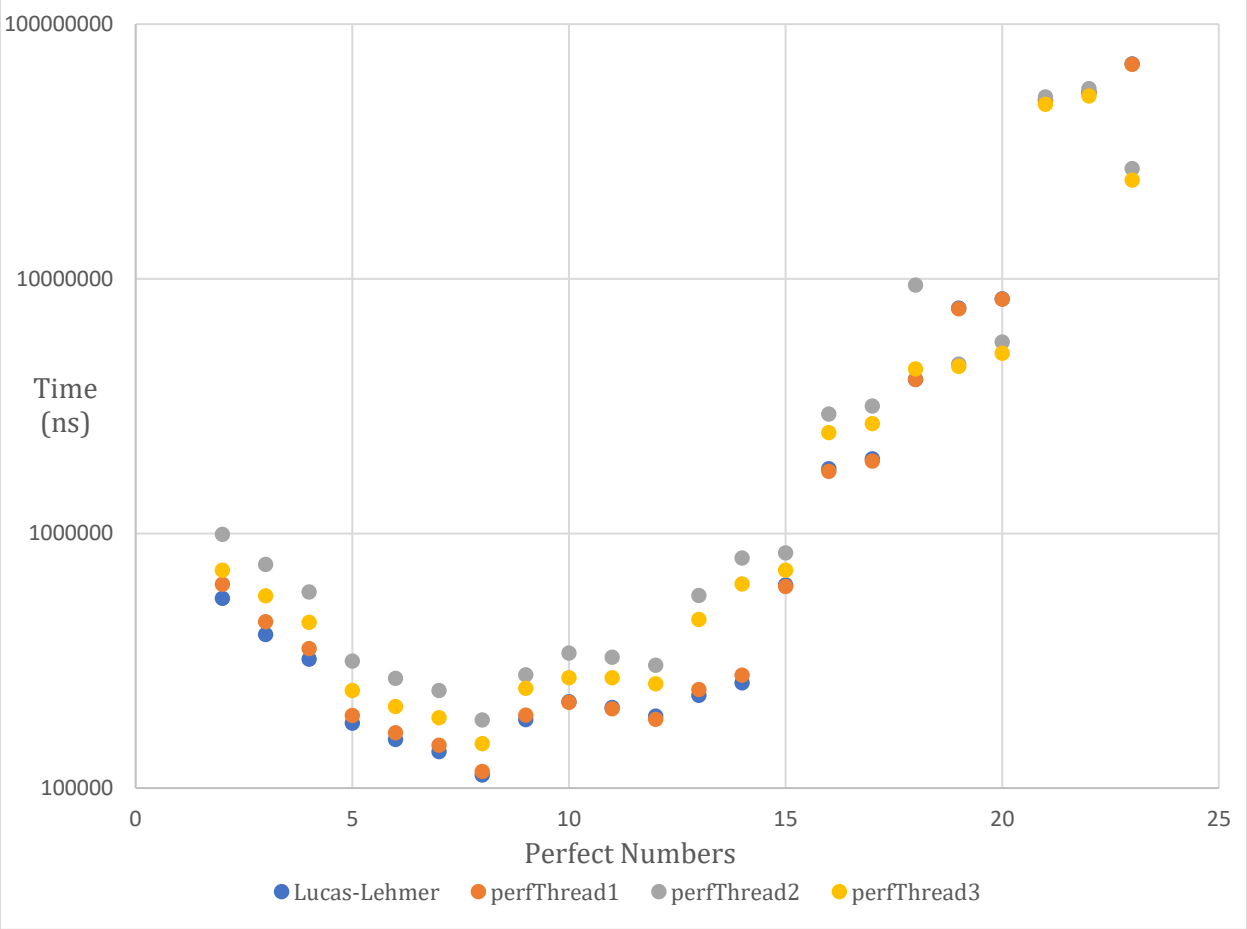

Figure 6. Time per digit in nanoseconds of the ordered perfect numbers for the LL method and multi-threading in Java.

Shown in Figure 6, there was a decreasing trend for each algorithm from the second perfect number to the eighth. The shared trend continued as the times per digit increased for two consecutive points and decreased for the 11th and 12th points. Notice that, up to the 12th perfect number, each method increased and decreased in unison by the nearly the same degree. A quartic polynomial with the equation $y = -417.49x^4 + 10247x^3 - 69333x^2 + 25922x + 870948$ fits that portion of the data for all four methods. That was conveyed by the given error value, $R^2 = 0.9815$. However, no consistent trend was shared between the methods for all 23 perfect numbers. From the 13th point onward, there was a general increasing trend, but the degree to which each time grew from the last varied between algorithms. There were a few noticeable outliers in the

remaining section. For example, the 18th point of perfThread2 was much larger than in the other algorithms. perfThread2 and perfThread3 increased by a small amount moving to the 15th perfect number while the LL method and perfThread1 experienced a large jump in time per digit. All four methods had similar times per digit for the 21st and 22nd perfect numbers. However, the times for perfThread2 and perfThread3 dropped immensely for the last perfect number.

perfThread1 is essentially the same algorithm as the LL method. The only differences are that it extends the Thread class and uses a separate Tester class to start the program. Both use only a single thread to compute all 23 perfect numbers. perfThread2 and perfThread3 differed from the LL method and perfThread1 by spacing out ranges of *n* between a set number of threads. The CPU used in the computer to implement the programs was an Intel Core i7-8850H. This CPU is quad core with hyperthreading giving it up to eight processable threads. Two variations of multi-threading were tested. perfThread3 assessed the effectiveness of assigning one thread per physical core vs perfThread2 which utilized hyperthreading. Multi-threading was faster for the larger perfect numbers since each thread was run at the same time. Therefore, all *n* did not need to be iterated through to reach a given *p*. In perfThread2, the range of the first thread included the most with 14 perfect numbers. There were four perfect numbers in the second thread range, two in both the third and fourth, and a single perfect number in each of the fifth, sixth, seventh, and eighth thread ranges. However, each of the last three threads were unable to find a perfect number within ten minutes of being run.

In perfThread2, the 23rd perfect number was found before the 21st and the 22nd was because it resided at the beginning of the range assigned to its thread. The 23rd perfect number was produced from p = 11,213 when the thread started at n = 10,000. The 21st number came from p = 9,689 when the thread began at n = 5,000. So, the fourth thread iterated through over 4,500 numbers to reach its first perfect number while the fifth processed less than 1,300. perfThread3 was the same with the 23rd perfect number existing at the beginning of the fourth thread with the range starting at n = 10,000. Even though the 23rd number was spaced the same distance into its range in both multi-threaded approaches, perfThread3 found the perfect number before perfThread2. This implies using four threads was more effective than eight. Using eight threads assigned two threads per core meaning resources were more spread out between threads. Therefore, the most optimal number of threads to use is the same number as the CPU's physical core count. perfThread3 made a small improvement by finding the 23rd perfect number 1.5 times faster than the LL method. However, the first 18 perfect numbers were found slower in both multi-threading approaches. For the LL method, perfThread2, and perfThread3, one thread was initialized at the same time with the first 14 numbers in its range. The only difference was how many additional threads were initiated. Since multi-threading spreads out the CPU's resources, thread1 was slowed down by the use of other threads in perfThread2 and perfThread3. The regular LL method did not have this problem. The slower runtimes for many perfect numbers and small improvement that existed only for the larger perfect numbers shows that multi-threading was not necessarily better than the LL method.

Delete Redundancy:

The second improvement of the LL method followed the same progression as the original version. For the purposes of this lab, the improvement was named "Delete Redundancy". A ratio between the two algorithms was calculated and shown in Table 7 to illustrate the differences in elapsed time. Recall time to find the first perfect number was inconsistent and out of trend due to

being calculated without the LL test. So, ignoring the outlier, the time to find the first seven perfect numbers was a small degree faster for Delete Redundancy than the LL method. The difference got smaller as more perfect numbers were produced. This was shown by the ratios between the two algorithms. The ratio started around 1.36 (rounding up) and generally decreased for every perfect number until it reached 1.20 at the seventh. The ratio between times continued to decrease for the remaining data but stayed near a value of 1.00 suggesting the improvement became less significant as *n* approached infinity. Figure 7 and Figure 8 on the following page solidified the trend made apparent by Table 7 illustrating the near equivalence of algorithms. Data for time per digit gave minimal insight to the algorithm's effectiveness. In general, it was clear that there was no significant improvement on the LL method. This shows that computing the primality of p is not the limiting factor to the LL method. Figure 8 was included without a table to display its values.

Table 7. Elapsed time in nanoseconds to find 23 perfect numbers with the Lucas-Lehmer method and Delete Redundancy in Java. Ratios between the two algorithms were included.

	Time to Find (ns)		
Perfect Number	**Lucas-Lehmer**	**Delete Redundancy**	**Ratio**
1	41100	2900	14.17241
2	1109050	818350	1.355227
3	1200600	885450	1.355921
4	1280450	943750	1.356768
5	1436600	1092400	1.315086
6	1549100	1237600	1.251697
7	1666950	1383950	1.204487
8	2132250	1936150	1.101283
9	6864900	6857650	1.001057
10	11765050	10808350	1.088515
11	13441600	12127900	1.10832
12	14754700	13147350	1.122257
13	72525350	72299300	1.003127
14	94657800	96150900	0.984471
15	482903850	475361700	1.015866
16	2376125550	2367185150	1.003777
17	2692964450	2684269650	1.003239
18	7779356550	7916093700	0.982727
19	19606232500	19834275950	0.988503
20	22178660600	22400707150	0.990088
21	2.92523E+11	2.96158E+11	0.987724
22	3.22596E+11	3.27614E+11	0.984685
23	4.70037E+11	4.79323E+11	0.980626

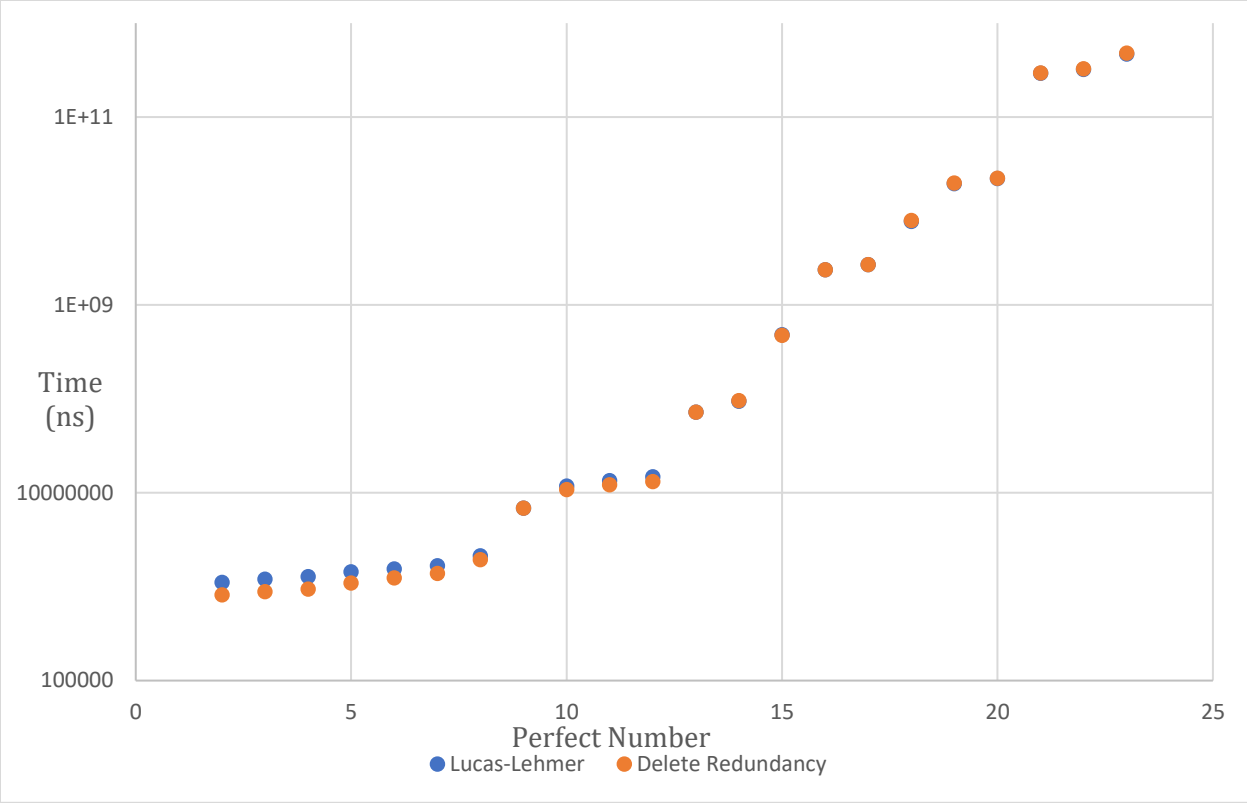

Figure 7. Elapsed time measured in nanoseconds for the LL and Delete Redundancy methods to find 23 perfect numbers in Java.

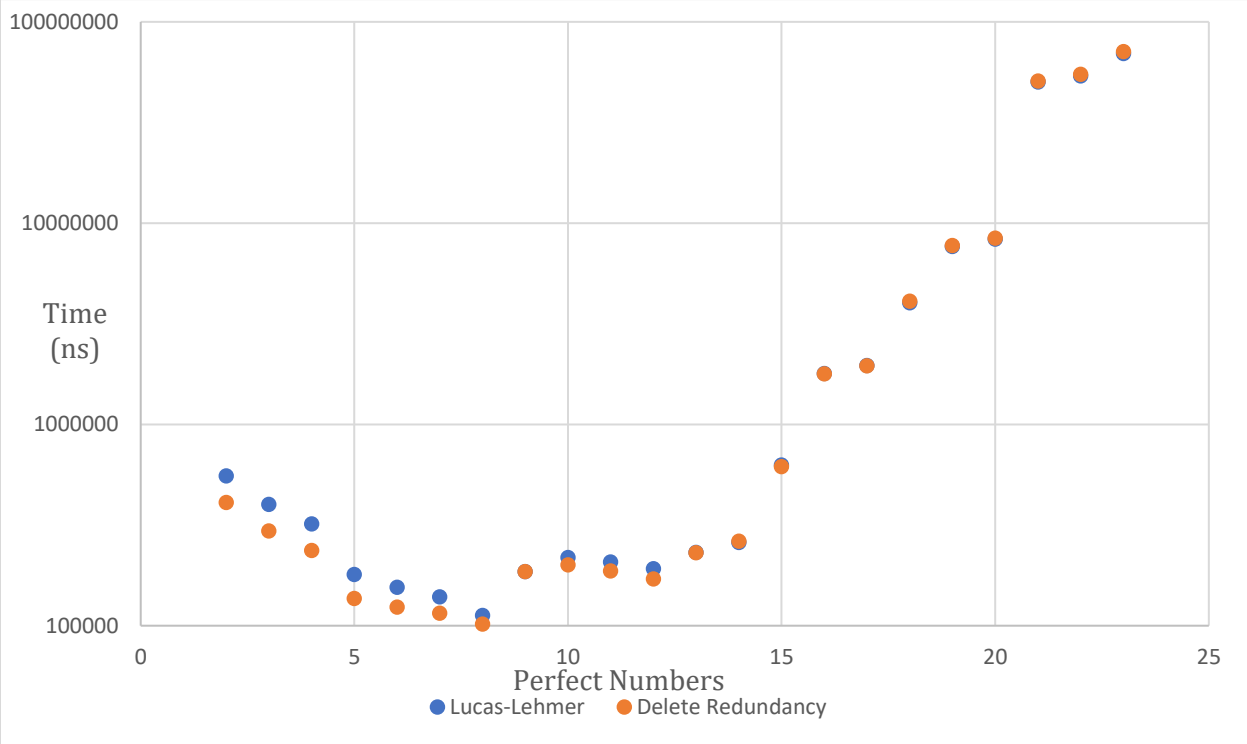

Figure 8. Time per digit in nanoseconds for the LL method and Delete Redundancy in Java.

Bit Level:

Similar to Delete Redundancy, computing numbers at the processor level did little to change the elapsed time of the LL method. The improvement was named "Bit Level" for easy reference. Opposite to Delete Redundancy, the elapsed time did not improve until the larger perfect numbers were reached. Before the 19th perfect number, all times were slower for the Bit level algorithm. It was only for the last five numbers that times were improved. The ratio fluctuated, increasing and decreasing almost randomly as time progressed signifying there was no consistent trend differing from the LL method. Additionally, the ratio lingered near 1.00 for most perfect numbers demonstrating the similarity between runtimes with 13 and 14 being obvious outliers. Using bitwise operations sped up the process for computing $(2^p - 1)$ and (2^{p-1}) but was unable to make any overall advancement from the LL method. This suggests that the algorithm is still limited by the LL test on Mersenne numbers. Figure 9 indicates the equivalent progression between algorithms. From the data, it is clear that Bit level made no significant improvement on the LL method.

Table 8. Elapsed time measured in nanoseconds to find 23 perfect numbers for the LL method and Bit Level in Java. Ratios were included between the two methods.

Perfect Number	Lucas-Lehmer	Bit level	Ratio
1	41100	42350	0.970484
2	1109050	1125275	0.985581
3	1200600	1295525	0.926729
4	1280450	1365700	0.937578
5	1436600	1511625	0.950368
6	1549100	1661175	0.932533
7	1666950	1824550	0.913623
8	2132250	2385625	0.893791
9	6864900	7989025	0.859291
10	11765050	12725000	0.924562
11	13441600	14853600	0.904939
12	14754700	16438075	0.897593
13	72525350	108097375	0.670926
14	94657800	163762975	0.578017
15	482903850	513132575	0.94109
16	2376125550	2833942475	0.838452
17	2692964450	3190725400	0.843998
18	7779356550	8233959375	0.944789
19	19606232500	1.5713E+10	1.247802
20	22178660600	1.7985E+10	1.233152
21	2.92523E+11	2.3024E+11	1.270524
22	3.22596E+11	3.0508E+11	1.05743
23	4.70037E+11	3.9582E+11	1.187501

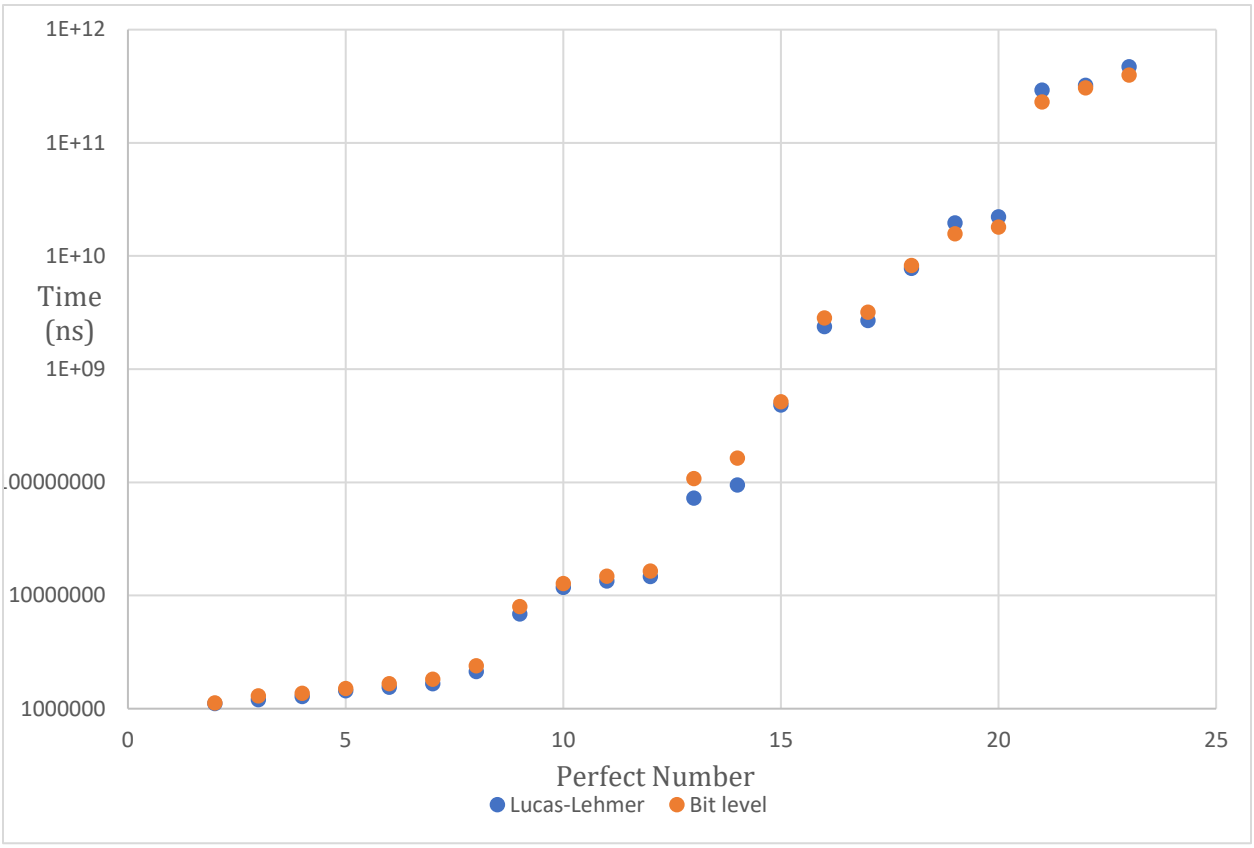

Figure 9. Elapsed time measured in nanoseconds for the LL method and Bit Level algorithm in Java.

Methods/Results (Python):

All methods were implemented in Python to compare the proficiency of algorithms in separate programming languages. Both were timed in nanoseconds, but data is not comparable cross-language due to different timing techniques used in Java vs Python. The data found similar results but signified that specific methods were less effective in Python. The Basic Algorithm, Method 2, LL method, Delete Redundancy, and Bit Level all shared the same trends as their equivalents in Java. One of the only major differences was that Method 2 was unable to produce a tenth perfect number within ten minutes of the last being found. The ratios between algorithms for each perfect number were comparable in Python and Java. The runtimes of Delete Redundancy and Bit Level never improved nor worsened by a significant degree from the LL method. This demonstrates that, similarly to the implementations in Java, the LL primality test was the limiting factor of the algorithms. Table 9 shows the runtimes from Python that revealed no distinct difference in trend from their Java equivalents. Rather it was the multi-threading approach that differed greatly from its version in Java.

Table 9. Elapsed time measured in nanoseconds for the Basic Algorithm, Method 2, LL method, Delete Redundancy, and Bit level to produced perfect numbers in Python.

	Time to Find (ns)				
Perfect Number	**Basic Algorithm**	**Method 2**	**Lucas-Lehmer**	**Delete Redundancy**	**Bit level**
1	21900	15500	15200	6900	16600
2	54700	32400	58200	36800	45000
3	2803100	40300	71700	48000	59300
4	880562600	47200	84000	58800	72000
5		61800	101500	71500	90700
6		84200	116600	83800	107100
7		123200	130200	95600	121800
8		4059300	164600	121600	161200
9		1.61266E+11	262900	198000	283500
10			397100	310000	452500
11			526200	424900	612500
12			648500	533100	762100
13			18591400	16716000	19417100
14			28627200	25895900	30001300
15			282628900	280799400	304729200
16			1821106100	1793633000	1974888800
17			2094658100	2066354200	2280483600
18			7173451400	7147613700	7784977400
19			20519417200	20440992300	22131450400
20			23445783900	23338992500	25238675700
21			4.32201E+11	4.25735E+11	4.58228E+11
22			4.80123E+11	4.72143E+11	5.08998E+11
23			7.36588E+11	7.27292E+11	7.79968E+11

Multi-Threading:

In Java, multi-threading was found to be an effective improvement on the LL method. However, the opposite was true in Python. perfThread1 was not programmed in Python since it was found to be almost an exact replica of the LL method. In Java, all perfThread methods produced the full 23 perfect numbers. However, the two implementations in Python failed to compute all 23 within ten minutes of the last number being found. perfThread2 found 20 numbers in total while perfThread3 found 21. It is important to note that, similar to Java, perfect numbers in perfThread3 were computed out of order. The 19th perfect number was found before the 18th while the 23rd produced without the 21st and 22nd ever being computed within the ten minutes. The method was ineffective since Python does not actually allow for true multi-threading to be implemented. This is due to the Global Interpreter Lock (GIL) which limits one thread to being executed at any given

time. The GIL moves between threads so that multiple threads are being utilized, but not concurrently. Evidently from Table 10 his just slows down the process and makes the algorithm less efficient.

Table 10. Elapsed time measured in nanoseconds for the LL method and multi-threading techniques implemented in Python.

Perfect Number	Lucas-Lehmer	perfThread2	perfThread3
1	15200	14400	14600
2	58200	144000	5800800
3	71700	13049800	17768600
4	84000	13099100	29886500
5	101500	44890500	29911200
6	116600	44946100	29927700
7	130200	166374300	29942500
8	164600	166456200	53741400
9	262900	214357100	100766700
10	397100	214574900	100909700
11	526200	14772500	101045600
12	648500	214971600	101176400
13	18591400	280184900	127672200
14	28627200	329847100	142296100
15	282628900	1189773900	552126200
16	1821106100	12417846000	5014653500
17	2094658100	14481951800	5904845600
18	7173451400	44556834000	15229025200
19	20519417200	63155408000	21228110000
20	23445783900	83336497900	23876388200
21	4.32201E+11		5.3343E+11
22	4.80123E+11		
23	7.36588E+11		

Conclusion:

Adding the Lucas-Lehmer primality test to method two made the largest improvement of all the algorithms. This demonstrated the limiting factor of method two was the basic primality test on Mersenne numbers. Due to the exponentially increasing size of Mersenne numbers, needing to compute trillions of modulus functions became a problem quickly and restricted the effectiveness of the algorithm. Once the Lucas-Lehmer method was found to make such large a difference, no other improvement became nearly as successful. Multi-threading improved runtimes by a small degree for the larger perfect numbers but only because the given threads range began at a higher *n* and each thread was run concurrently. Times were slower for the first 18 perfect numbers since resources were divided among cores. Overall, multi-threading did not provide a major improvement on the Lucas-Lehmer method. Delete Redundancy produced runtimes equivalent to the Lucas-Lehmer method. Therefore, the primality test on the basic *p*'s was not a significant

contributor to the slow runtimes of the Lucas-Lehmer method. Lastly, Mersenne numbers and their multipliers were computed by shifting and setting bits instead of using multiplication. This was also found to make no major difference from its original version. Since no method made sizable improvement on the Lucas-Lehmer method, it becomes apparent that the limiting factor to computing perfect numbers was the Lucas-Lehmer test's capacity to determine the primality of Mersenne numbers. Perhaps a new and more innovative primality test, like the AKS primality test, could make a substantial improvement on the Lucas-Lehmer method.