Michelle Cheng

MA357 Paper 2

5/1/2020

# Factoring of Numbers

The issue of factoring numbers has always been relevant in everyone’s daily lives. It’s taught to us as children when we learn to share and divide candies into equal portions, used in architecture and crafts to make sure components fit perfectly into the whole, and now, with the World Wide Web touching almost every part of our lives, we rely greatly on the difficulty of factoring large numbers in many cryptographic algorithms to protect our privacy. In this paper, we’ll explore six different factoring methods—trial division, Fermat factoring, Fermat factoring combined with trial division, Fermat factoring combined with the sieve method, Fermat factoring combined with sieve method and trial division, and pollard’s rho factoring method—by using them to factor four kinds of interesting prime-generating numbers—Mersenne numbers, Repunits, Fermat numbers, and Cullen numbers—and analyze the results based on time efficiency and accuracy.

## Methods:

For the rest of this paper, the variable will refer to the number being factored and the worst-case scenario will always be when is prime.

Trial Division

This is a brute force method that checks for a non-trivial prime factor of . If one exists, we know it must be less than. Implementation of the algorithm is simple:

1. Factor out all factors of 2, save to list of factors, let
2. Factor out odd factor from , save to list of factors, replacing by
3. Increase d to next odd factor
4. Repeat steps 2 and 3 until
5. If does not equal one, add to list of factors
6. Return list of factors

If the returned list of factors has only one element, , then we know it is prime. Otherwise, it must be composite. Since in the worst-case scenario, this algorithm checks all the primes up to , according to the prime counting function we check approximately factors. Rounding up, the worst case time complexity for factoring is approximately Ο().

Fermat Factoring

This method is based on the fact that every odd number can be represented by the difference between two squares. In the case of a prime, the only possible factors are and , so with , and . With this algorithm, we start with and work our way up until we either find non-trivial roots and or and and we conclude that is prime. Implementation of the algorithm:

1. Let
2. If is a square, set and return factors
3. If not, let , ,
4. If , go back to step 1
5. If , return that the algorithm has failed to factor n

If the algorithm returns a list with 1 as the first factor, then we know the number is prime. If the first number is not 1, then we know it’s composite. If the algorithm returns a failure, then is probably not large enough. If is already large, the method is probably not the most time-effective way to factor the number. In the worst case scenario, the factors we find are , so works from up to , so rounding up, would guarantee that the algorithm does not result in failure. However, since we have to do multiple squaring and square-rooting operations in each iteration of this method, it would result in a worst-case time complexity of Ο() which is actually much slower than trial division.

Fermat Factoring with Trial Division

Instead of trying to find factors by running the Fermat factoring algorithm times, if we set to a lower number like 5000, then use trial division from there, we’d be able to use trial division to try to find a factor in a smaller scope. More specifically, we’d only have to test primes up to since if there was a prime factor between up to , the Fermat factoring algorithm would’ve already found it.

Fermat Factoring with a Sieve

Another way to increase the efficiency of the Fermat factoring algorithm is to decrease the amount of times the power function is called since the power function takes Ο() each time it is called. We take into consideration the values of that can be a square. In my case, I had , so . I calculate, then figured out the list of possible and resulting list of . Then, for each value of , I check if its modded value is in . If it is, we proceed with step 2 of the Fermat factoring algorithm, if it isn’t, we increase the value of by one and check again. However, while python modulo is Ο(1) for small numbers, it’s Ο() for large numbers, so we’ll have the greatest improvements in time efficiency if we are working with small values of . Therefore the performance of factoring with a sieve would probably work much better when used in combination with trial division just so that we avoid making the value of too large.

Pollard’s Rho Algorithm

This method is based on related congruence classes and the birthday paradox. Given a composite number and a pseudorandom sequence generating function , letting , we get a sequence that is related to the sequence , which means the values will eventually repeat with some . The birthday paradox says that a repeat will occur around . The implementation is:

1. Let
2. While , let and ,
3. If , return false
4. Otherwise, return

With this algorithm, different starting values of x and y may result in different factors being found. Sometimes, a bad starting value may result in the algorithm failing to find an existing non-trivial factor. I simply set my code up so that the algorithm uses a random starting value between 1 and 9. In terms of efficiency, this algorithm is quite fast. Since I kept the starting values small and the variables and are modded in every pass of the algorithm, the values and grow but is kept smaller than . The gcd algorithm uses the binary Euclidean algorithm, so the time complexity is Ο(). Therefore the time complexity of this algorithm is Ο() which is only slightly slower than trial division in the worst case scenario.

Numbers Used:

* Mersenne numbers
* Mk = 2k – 1
* k = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53]
* Repunits
* Rk = 11…1 with k digits = (10k – 1)/9
* k = [2, 3, 5, 7, 11, 13, 17, 19]
* Fermat Numbers
* Fk = 22^n + 1
* k = [2, 3, 4, 5]
* Cullen Numbers
* Ck = n\*2n + 1
* k = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31]

These numbers were picked because they tend to generate primes and so would make for interesting numbers to factor. The list of k values were mostly picked from a list of primes and simply stopped when it took longer than half an hour to factor the number. Most of the time, the list had to stop due to the Fermat factorization algorithm taking too long.

## Analysis of Results

Results of all the numbers tested were combined. For numbers less than 10 million, the factorization times of Fermat factorization were still manageable so time taken for all the methods are put on the same plot, but past 10 million, Fermat factorization took a significantly longer time so the two methods that relied mainly on Fermat factorization were put on separate plots. Plots are further broken down into three subplots in order to better explore the trends of time efficiency as large numbers grew too far spaced apart.

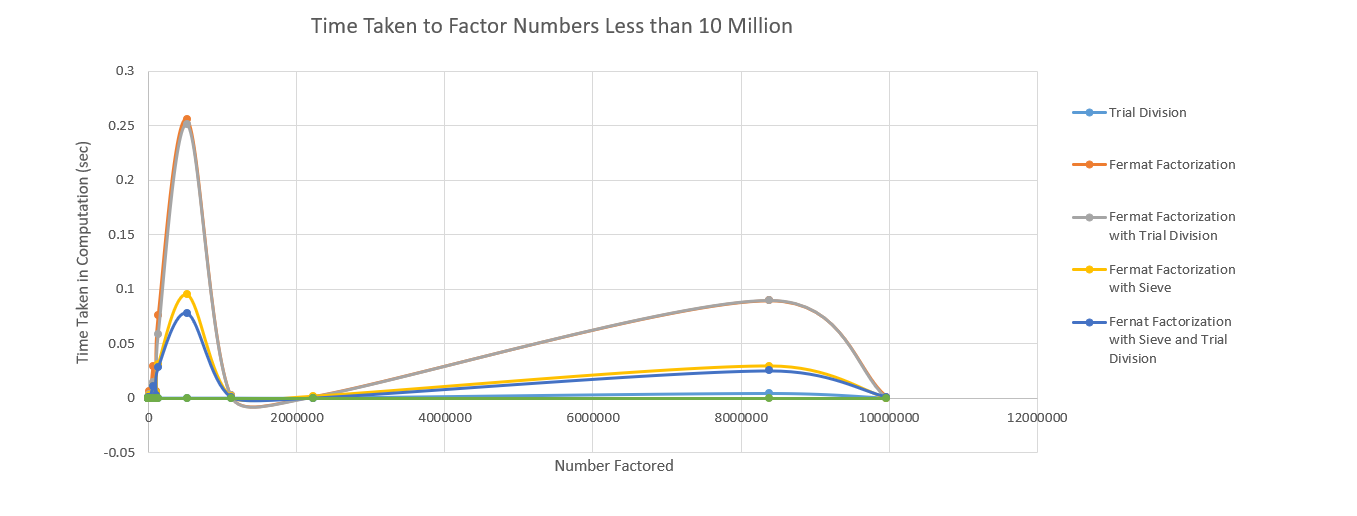
Time Efficiency Analysis

Figure 1: Plot of Numbers Less than 10 Million

For numbers less than 10 million, Pollard’s Rho took consistently around 0 seconds and trial division took mostly 0 seconds with a few numbers taking from 0.0009 to 0.01 seconds. As shown by the chart, with the value 524287, Fermat factorization took slightly longer than Fermat factorization with trial division, but for these numbers, they tend to take around the same amount of time and take the most time out of all the other algorithms. This is likely because I had for Fermat factorization set to and for Fermat factorization with trial division set to 999,999, so for these numbers, Fermat factorization likely finds a result before hitting 999,999 iterations, hence trial division isn’t being used much, if at all. It’s also a nice note that, as we predicted, Fermat factorization with sieve works much faster than Fermat factorization and adding trial division onto the sieve method increases the speed a bit more, at least for these numbers.

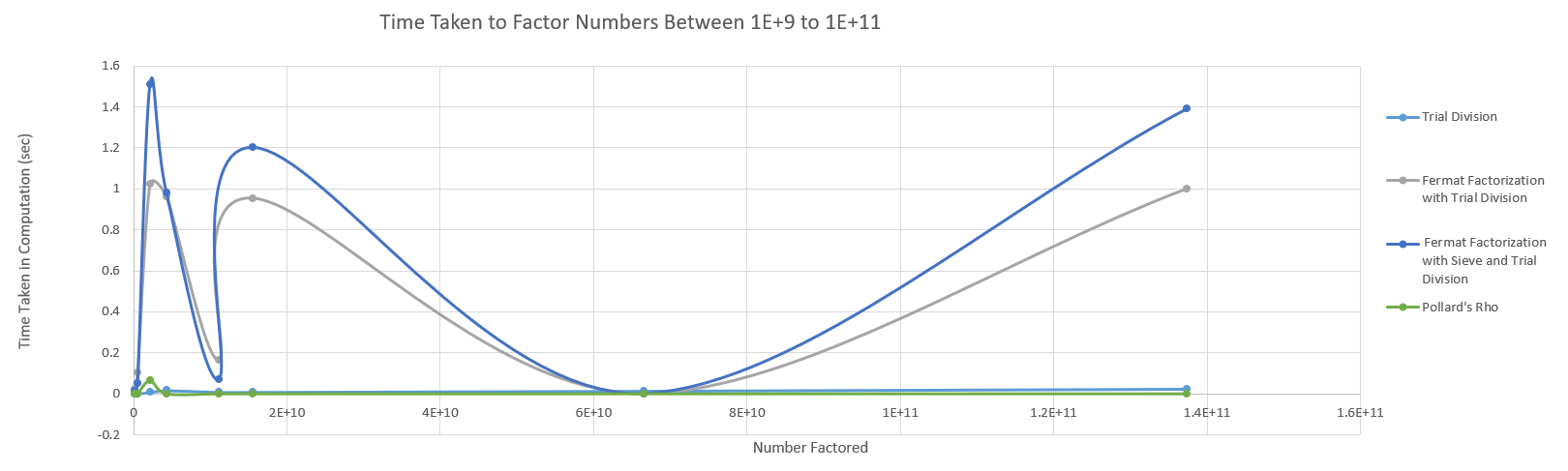


Figure 2: Plot of Numbers Between 1E+9 to 1E+11 with Small Factorization Times

With this range of numbers, time required for trial factorization remains close to 0, but now range around 0.009 seconds while Pollard’s rho still takes around 0 seconds with the one exception of 2147483647 taking 0.063 seconds and still being correctly classified as prime. With these numbers, Fermat factorization with trial division works faster than Fermat factorization with sieve and trial division, but they share similar trends.

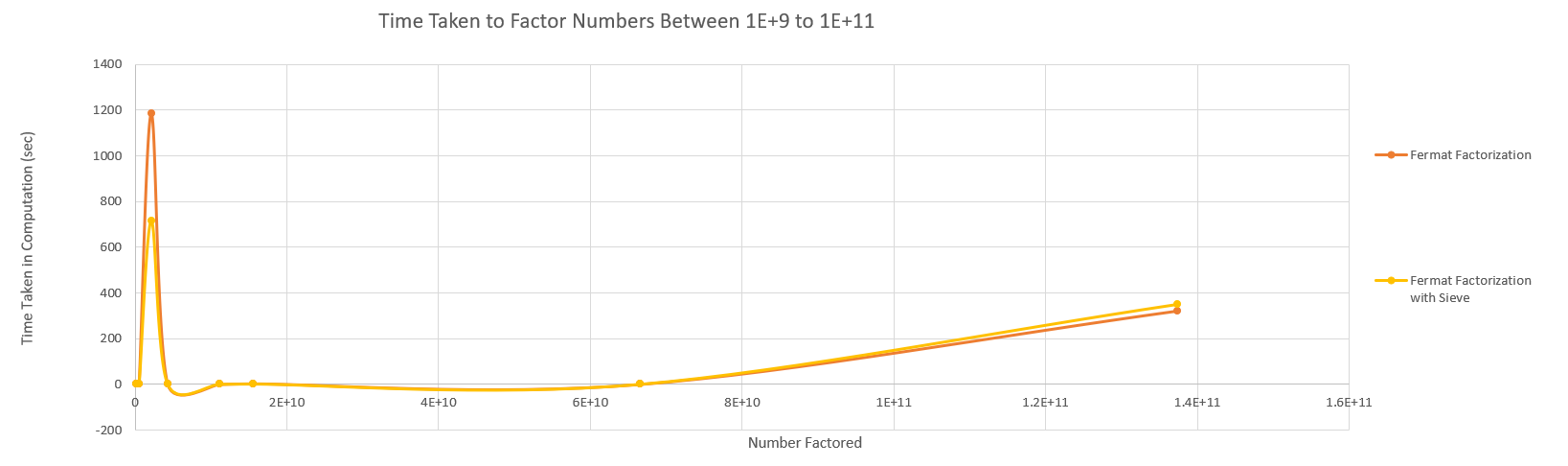


Figure 3: Plot of Numbers Between 1E+9 to 1E+11 with Large Factorization Times

When we use Fermat factorization without breaking at a constant point and shifting to trial division, it takes significantly longer. Figure 2 had been on the scale of 0 to 1.6 seconds but Figure 3 is on the scale of 0 to 1400 seconds! However, this plot seems to confirm our previous apprehensions. Initially, the sieve method is faster than the non-sieve method at factoring 2147483647. It cuts the time by almost half, but as the numbers we factor increase in value, it takes about the same amount of time for both methods to factor until for the last point in this plot, 1.37E11, the sieve method takes longer than not having the sieve functionality added. At this point, the value of is probably so large that modding it is causing a significant overhead.

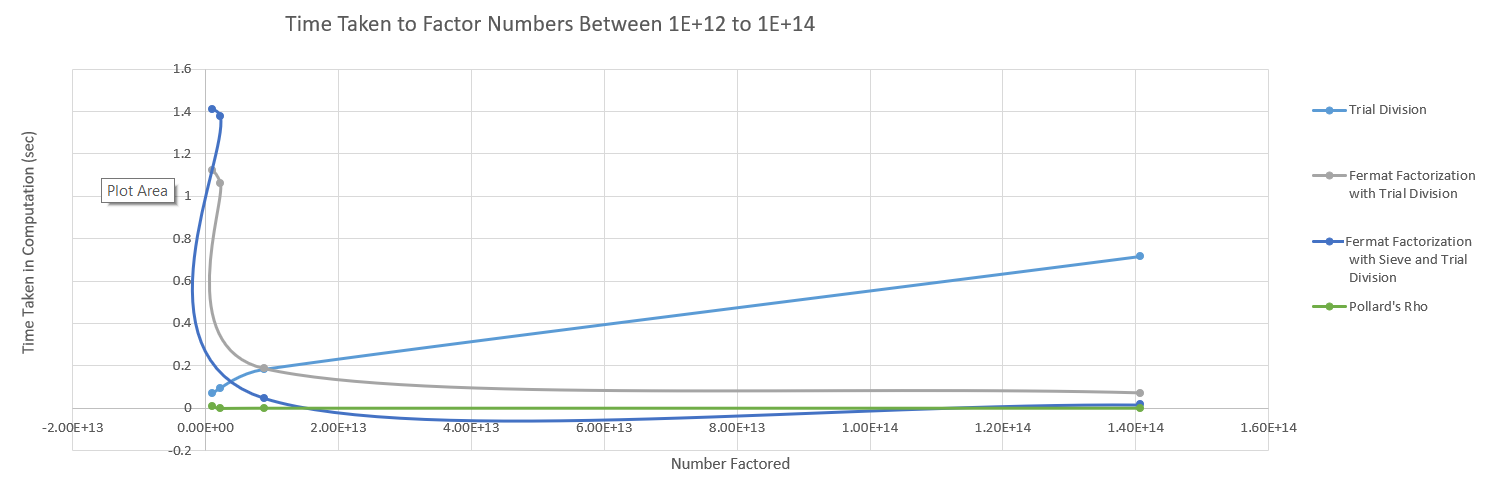


Figure 4: Plot of Numbers Between 1E+12 to 1E+14 with Small Factorization Times

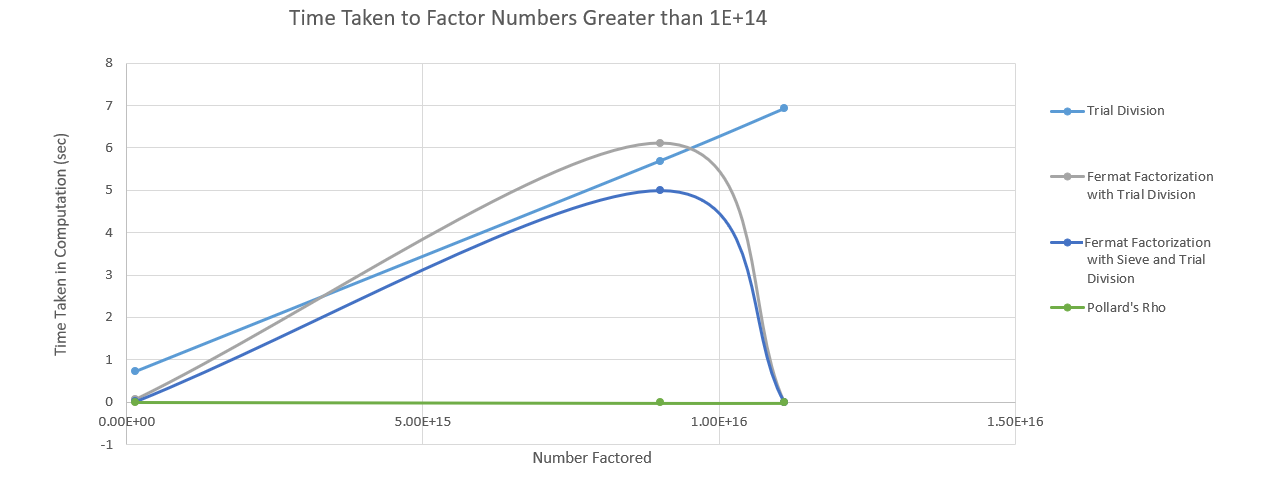


Figure 5: Plot of Numbers Greater than 1E+14 with Small Factorization Times

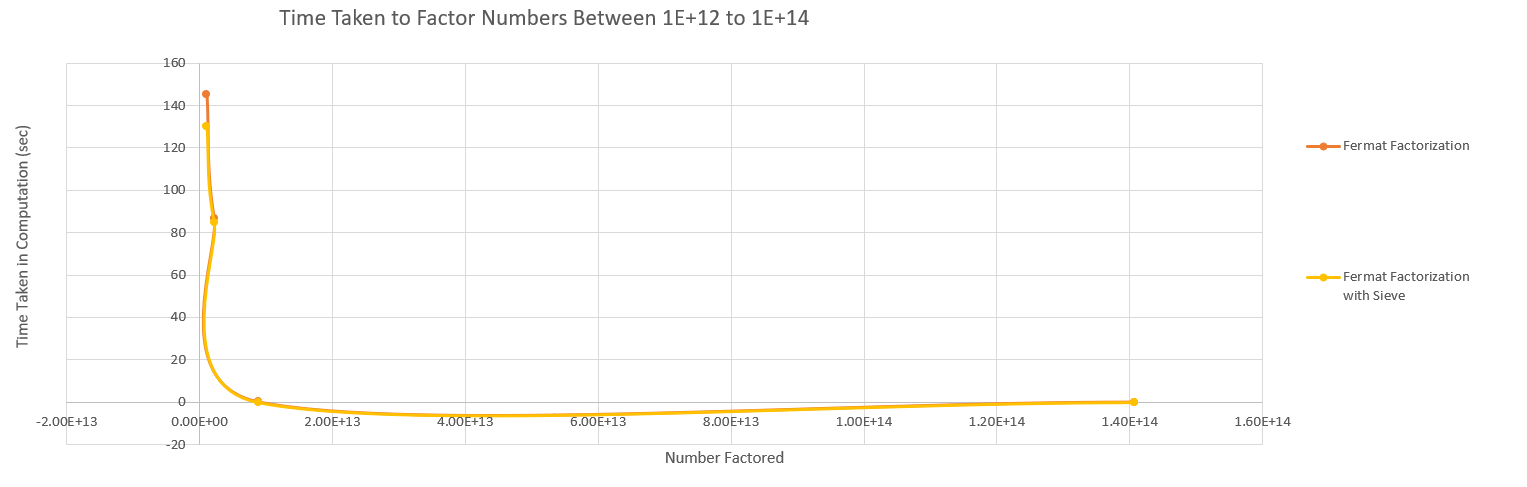
Figure 4 and Figure 5 only encompasses seven numbers between them, but because the values are too spread out, there was a need to make two separate plots in order to see the trends clearly. We can see from both figures that the time trial division is taking to factor is steadily increasing and remains significantly above the other methods for most of the numbers displayed. Pollard’s Rho continues to remain near 0, and the two Fermat factorization methods combined with trial divisions continue to share similar trends with the sieve being slightly faster. It is nice to see that at around 8.8E+12, Fermat factorization combined with trial division finally grows to work faster than the brute force method of trial division.

Figure 6: Plot of Numbers Between 1E+12 to 1E+14 with Large Factorization Times

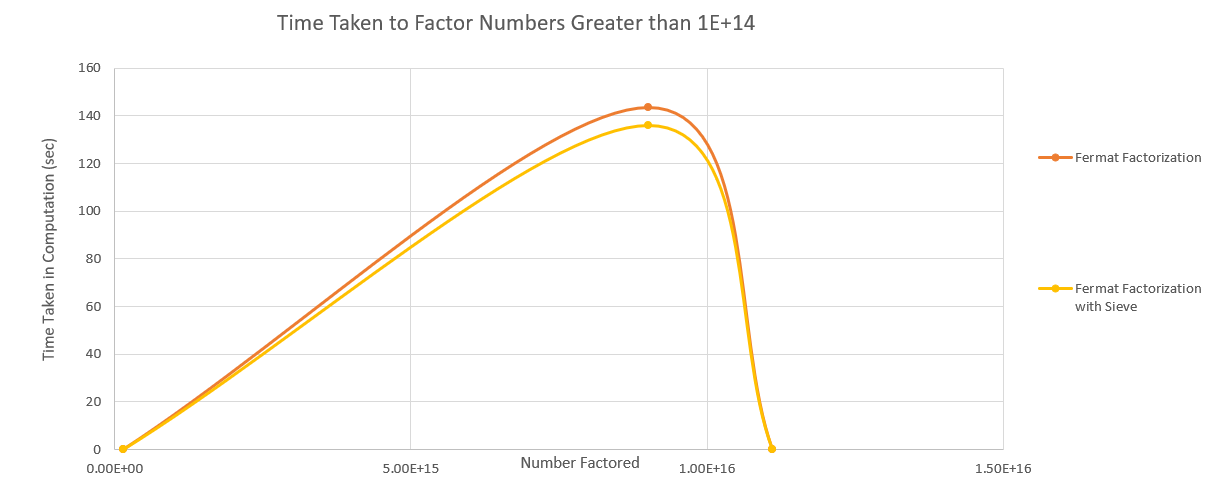


Figure 7: Plot of Numbers Greater than 1E+14 with Large Factorization Times

As we would expect, with extremely large values, the sieve method doesn’t improve the time of Fermat factorization by much since the time required to mod a large value is comparable to the time it would take to square the value.

Overall, in terms of time efficiency, Pollard’s Rho beats all other methods out by a mile. Trial factorization also works impressively well for numbers less than 1E+12. After that, the data we have available suggest that Fermat’s factorization method combined with the sieve method and trial factorization would work faster than all methods presented here other than Pollard’s Rho.

Accuracy Analysis

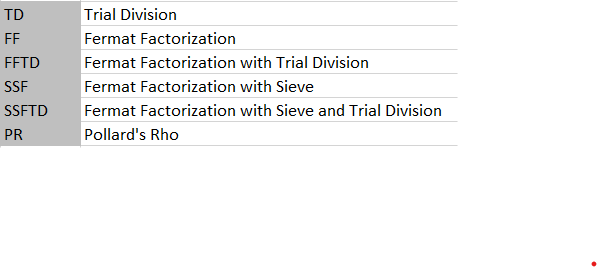


Figure 8: Abbreviation Key



Figure 9: Accuracy Table

As we would expect, accuracy for trial division is 1. It perfectly classifies what is prime and what is not since it tests all possible non-trivial prime factors. Fermat factoring is the least accurate out of all the methods we tested. I think the basis for the inaccuracy is that it has difficulty finding factors for small numbers. All the Fermat factoring methods misclassified 9 and 25 as prime, resulting in the 95% accuracy. Pollard’s Rho was surprisingly accurate. My apprehension with the fact that it may often return a failure due to a bad starting value was completely unfounded as the result after one run with a randomly picked starting value was very accurate. The only value it had misclassified as prime is 25, but perhaps a repeated trial with a different starting value would’ve fixed that.

## Conclusion

In this paper, I was mainly concerned with analyzing all six methods together in order to get an overall view of the strengths and weaknesses of the algorithms in relation to each other. The conclusion I’ve come to at this point is that Pollard’s Rho is probably the most efficient in terms of both time and accuracy at factoring large numbers. However, further analysis comparing trial division, Pollard’s Rho, and Fermat factoring with sieve and trial division may paint a different picture.

Further Explorations

1. Compare Fermat factoring with sieve and trial division against Fermat factoring with trial division to see if the gap between their time efficiencies eventually grow significantly large
2. Try Pollard’s Rho with static starting values to assess if there are certain starting values that do better than others
3. Find numbers which require multiple trials of Pollard’s Rho
4. Compare trial division, Pollard’s Rho, and Fermat factoring with sieve and trial division for numbers larger than the ones that have been explored in this paper

Lastly, to link back to the idea of cybersecurity based on the difficulty of factoring large numbers, RSA keys are generally 1,024 or 2,048 or 4,096 bits. The largest number factored in this paper was 9.01E+15, which is only 54.0 bits and it had taken over 2 minutes to factor with Fermat factoring. With trial division, it had increased from taking 0.71 seconds to factor 1.41E+14 to taking 5.69 seconds to factor 9.01E+15. Increase of a single bit doubles the size of a number, so it’s easy to imagine how time-intensive of a task it must be to factor numbers that are 1,024 bits and above.