**How accurate and efficient can computers be, aided by mathematics, in calculating the amount of primes up to a given limit?**

**By Alex Austin**

**Contents**

**Section 1:**

1.1-Abstract

1.2-Introduction

**Section 2:**

2.1-Defining mathematical and algorithmic methods

2.2-Mathematical methods

2.3-Algorithmic methods

**Section 3:**

3.1-Discussion: Parameters of investigation

3.2-Ease of Implementation

3.3-Results of Implementation

3.4-Speed of Methods

3.4-Conclusion

**Section 4:**

4.1-Evaluation

**Introduction**

Prime Numbers themselves are a simple concept; they are only divisible by themselves and one. Despite this mathematicians have spent centuries trying to gain a superior understanding of the distribution and patterns in which they form. There is evidence of the study of prime numbers dating back to the time of the ancient Egyptians, however the first explicit records of the study of prime numbers hails from the Ancient Greeks. Euclid showed the infinitude of primes and the fundamental theorem of arithmetic within ‘Euclid’s Elements’ (circa 300 BC), the former a fundamental proof in the understanding of the distribution of prime numbers. In 1640 along came Pierre De Fermat with ‘Fermat’s Little Theorem’ which is the basis of the ‘Fermat primality test’. Other influential mathematicians also contributed such as Euler; who showed that the infinite series 1/2 + 1/3 + 1/5 + 1/7.... is divergent. And at the beginning of the 19th century, Legendre and Gauss independently hypothesised that as x tends to infinity, the amount of primes up to x is asymptotic to x/ln(x). Towards the end of the 19th century further primality tests where designed in order to cope with much greater numbers for instance the Lucas ‘primality test’.

Before entering the real question at hand, I feel it is necessary to demonstrate that there is an infinite amount of prime numbers. This is for nothing more than to prove that there can be prime numbers of real significant magnitude for later parts of the project. The generic proof is that from ‘Euclid’s Elements’:

*Suppose there is only a finite amount of primes, this amount is called n. We denote them by . Now we can form a new number:*

*\* \*\*...\**

*As p is greater than any of the primes, it doesn’t equal one of them. Since p >, p cannot be prime. Thus is must be divisible by one of our finitely many primes, say (with 1 ≤ m ≤ n). But what actually happens when we divide p by is that we get remainder 1. This is a contradiction to our original assumption that there must be finitely many prime, therefore this assumption must be false. Thus there must be infinitely many primes. By Contradiction.*

Mathematics and computer science have been inherently linked since the formations of modern computer science in the 1940s. Mathematics underpins much of the logic and algorithmic thinking required for computer science, particularly within functional programming. Computations have been used to calculate and simulate across all areas of mathematics, from statistical data, to physical collisions, to quantum mechanics, to Number Theory. Turing used the Manchester Mark 1 Electronic Computer to verify the first 1104 non-trivial zeros of the Riemann Zeta Function, the ones with 0 < Im(s) < 1540. These computations by Turing were an exceptional feat; however they raise crucial questions about the role of computers within mathematics. Can we ever develop computers than can prove mathematics? Are computers the most efficient method of calculating such complex mathematics?

In this project I intend to evaluate the most efficient method in order to calculate, by computation, the number of primes up to a given limit. Moreover I intend to evaluate whether mathematical methods are more or less useful than algorithmic methods, in terms of difficulty, speed and accuracy.

**SECTION 2**

**2.1- Mathematical and Algorithmic Methods**

What is a mathematical method? I am going to define a mathematical method as a singular or non-iterative calculation used to calculate an exact answer or approximation to the amount of primes up to a given limit.

What is an algorithmic method? An algorithmic method is a method that uses a primality test and then runs iterations for every integer until the maximum limit is reach. In general the methods within this section are much less mathematical than in the mathematical methods. These methods take a much more logic based approach.

**2.2- Mathematical Methods**

**Legendre’s Constant**

For this specific project does not mean \* x but the number of primes up to a given limit (the Prime Counting function). In 1808 Adrien-Marie Legendre released a paper which stated that was particularly close to , where x is the limit to which primes is counted to.[1]. Or in its more common mathematical notation (In actual fact it is an asymptotic form but it easier to say it is equal to :

This can be rearranged to: ) = B

B is referred to as Legendre’s constant. It is not entirely known whether Legendre made a guess upon the value of B or whether he noticed that that the first 100, 000 elements within the sequence appear to converge to a value which is approximately 1.08366. [8]

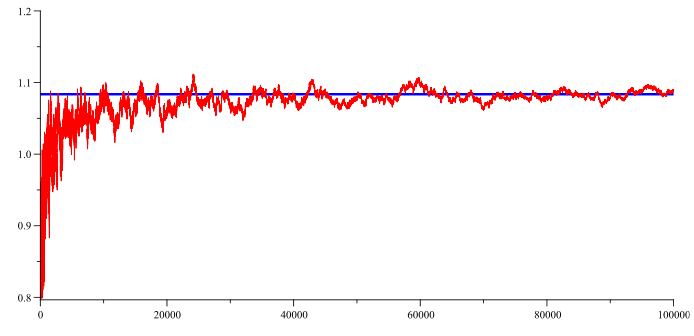


Figure 1- Shows the first 100 000 elements of the sequence, red line shows the elements and blue line shows Legendre’s constant [9]

However, in 1849 Pafnuty Chebyshev (Russian mathematician 1821-1894) proved that if such a limit exists (B), it must be equal to 1. A more straightforward proof that B must be equal to 1 was formed by János Pintz (Hungarian mathematician 1950-) in 1980. Despite the methods failings it is still an important part of the history of Prime Number Theorem so should not be discounted.

**Methods of Gauss**

In Germany, 1792, a 15 year old maths prodigy going by the name of Carl Friedrich Gauss noticed, whilst looking through his logarithmic tables, that there was a relationship between x and ln(x). He saw that this relationship was related to the primes.[1] Gauss noted on his logarithmic tables:

Gauss did not publish this, most probably as he did not have proof. This approximation only came out in a letter to the Johann Encke (a German astronomer) in 1849.

In 1838, after a string of communications Gauss and Peter Gustav Lejeune Dirichlet (a German mathematician credited with founding the field of analytic number theory) formed a better approximation for . tends to give an overestimate for when is less than (Skewe’s number), although, this number has since been greatly reduced to by Bays and Hudson [27]. The method boils down to the logarithmic integral function.[1]

This integral most probably confirms that the density of primes should be approximately 1/ln t [11]. Moreover has the asymptotic series about of:

And taking these first three terms has been shown to come up with a better estimate than Gauss’s initial . [4]

The nature of Li(x) allows you to be able to apply the Mean Value Theorem to it in order to find a lower bound for it.[10]

*Mean Value Theorem: Let F be a continuous function on [a, b] and differentiable on (a, b). There exists a C (a, b) s.t.* [10]

There exists C s.t.

[As and Li(2) = 0 ( = 0)]

This result is often used as neat method to check the validity of an answer given the great size that Li(X) can reach when a large limit is set.

**Riemann Hypothesis**

Bernhard Riemann inspected the properties and behaviour of the function:

This later became known as the Riemann Zeta function, where s is a complex number. The trival zeros of the zeta function are when = 0 and Re(s) 0, these zeros lie on the negative even integers. There are no zeros in the region R(s) since the series is always diverges in this region.[15] The more interesting and relevant zeros of the zeta function lie in the ‘critical strip’ situated in the region 0 < Re(s) < 1. So far we have managed to prove that there are infinite zeros that lie on the line ½ + it (‘Critical Line’), no one else has ever found a non-trivial zero that lies within the strip that is not on the ‘Critical Line’. This is the Riemann Hypothesis.[14]

If the Riemann Hypothesis were to be true, then it would have huge consequences in size of error in prime number theorem. The theorem states that for large, the ratio of to gets much closer to 1. However this is the difference relative to , but in actual fact the difference between and becomes greater and greater as increases. Computer simulations show that the error term is approximately proportional to . In 1901, Swedish mathematician Helge Von Koch constructed a proof such that the Riemann Hypothesis is logically equivalent to:

Where and the vertical lines indicate the absolute value (the difference is multiplied by 1 in order to give a positive value, alternatively the modulus). This gives currently the best bound on the error difference between and .[1] However this method falls down on the fact that it cannot be proven to be true without proving the Riemann Hypothesis.

In 1859, Bernhard Riemann wrote a paper entitled ‘Ueber die Anzahl der Primzahlen unter einer gegebenen Größe’ (On the Number of Prime Numbers less than a Given Quantity). In which Riemann gave an explicit formula for [29]:

Where:

This function gives a remarkably reasonable approximation of , the table below shows the accuracy of the Riemann R function. The function is however, deceivingly simple by look but in reality the use of the non-trivial zeros of the zeta function make the maths particularly challenging.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Overestimate | Difference (%) |
|  |  | 1 | 25 |
|  | 26 | 1 | 4 |
|  | 168 | 0 | 0.0000000 |
|  | 1227 | -2 | 0.1627339 |
|  | 9587 | -5 | 0.0521268 |
|  | 78527 | 29 | 0.0369436 |
|  | 664667 | 88 | 0.0132415 |
|  | 5761552 | 97 | 0.0016836 |
|  | 50847455 | -79 | 0.0001554 |
|  | 455050683 | -1828 | 0.0004017 |
|  | 4118052495 | -2318 | 0.0000562 |
|  | 37607910542 | -1476 | 0.0000039 |

Figure 2- Shows the values of R(x) for values of x up to 1,000,000,000,000[29].The percentage difference has also been included as a point for comparison.

**2.3- Algorithmic methods**

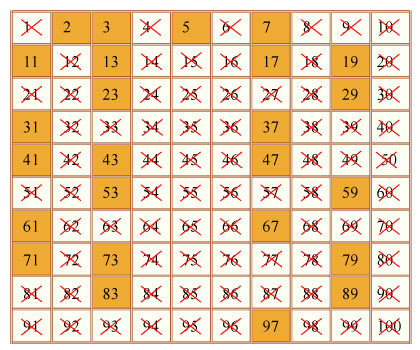
The algorithmic methods mentioned below are a selection of primality tests and other algorithms that can be used to tackle how many primes there are up to a given limit.

When asking Google for the definition of a primality test it returns:

*A primality test is an algorithm for determining whether an input number is prime. Unlike integer factorisation, primality tests do not generally give prime factors, only stating whether the input number is prime or not.*

**Sieve of Eratosthenes**

Primality testing first appeared in Ancient Greek texts, certainly a pattern forming here! Most probably the first method was conjured by Greek mathematician Eratosthenes (276 BC-194 BC) when he created the ‘Sieve of Eratosphenes’ algorithm. The algorithm is best described by the analogy:

*Every number represents a rock, where the shape of each rock is determined by its factors. Imagine that we have a magic bowl with adjustable holes in the bottom. The numbers (rocks) are then placed in the bowl and the adjustable holes are used to filter out the numbers that have a factor of 2, thus leaving all the numbers that do not have a factor of 2 in the bowl. This is then repeated for factors of 3, 5 etc until there is only prime numbers remaining.[16]*

Assuming the method is now housed in a table of length of the limit and one singular row, and if you define the first number in the list k. From the beginning of each round the method follows:

Figure 3-Shows the ’Sieve of Eratosthenes’ algorithm applied to a limit of 100, all of the primes in this range remain un-crossed.[17]

*-k is not a multiple of any of the numbers to the left otherwise it would’ve have been crossed off.*

*-We can now say that k must be prime as all of the numbers were written in order.*

*-Starting at k, move through the list in steps of k, crossing out multiples of k. This is because k + k = 2k etc, meaning that every number in which you will land on will be a composite number of the form n x k.*

**Trial Division**

The method centres on seeing whether a number, n, can be divided by any integer less than n. In order for this algorithm to be run at maximum efficient it requires the user to only select prime numbers less than n as factors. This is because if you have tested 2, you have tested every multiple of 2 and if you have tested 3, you have tested every multiple of 3 etc. If the algorithm finds one prime factor of n then it is prime and if more than one prime factors are found then it is a composite number.

**Fermat Primality Test**

Fermat’s Little Theorem states that if is a prime number and is a natural number, then:

Moreover, if does not divide , then there exists some smallest exponent s.t.

And that must divide . Thus:

This theorem is also sometimes entitled ‘Fermat’s Theorem’. Fermat’s Little Theorem is the basis of the Fermat primality Test.[18]. If testing for whether is prime, then it is necessary to pick a random that is not divisible by , in order to see whether the equality holds. If this equality holds then is prime and if not is composite.

*Example:* If we take to be 221, and randomly pick a value for in the interval 1 < < 221, let’s take = 38. Then:

Therefore we can conclude that 221 must be a prime number. However, it is often useful to check another value of . So let’s to be 24:

This proves that is in fact a composite number and that 38 was a ‘Fermat Liar’, but 24 is a ‘Fermat Witness for 221 being composite.[19]

As you can see these ‘Fermat Liars’ are a particular problem to the algorithm. But there are even greater problems for the method, Carmichael Numbers. Carmichael numbers are ‘Fermat pseudoprimes’, meaning that they will always pass the test as a prime regardless of which base () you choose. [20]The first Carmichael number is 561, this is clearly not a prime given its prime factorisation of 3\*11\*17.

**Solovay-Strassen Primality Test**

Firstly the test is based upon a proof constructed by Euler:

Where is the Legendre symbol (a multiplicative function with values 1, -1 and 0). The method relies on this congruence when n = .

The values that the symbol has are dependent upon the values of and (The odd prime numbers). The relationship between and determines which of the three values it has (1, -1 or 0). The three outcomes occur via these ways:

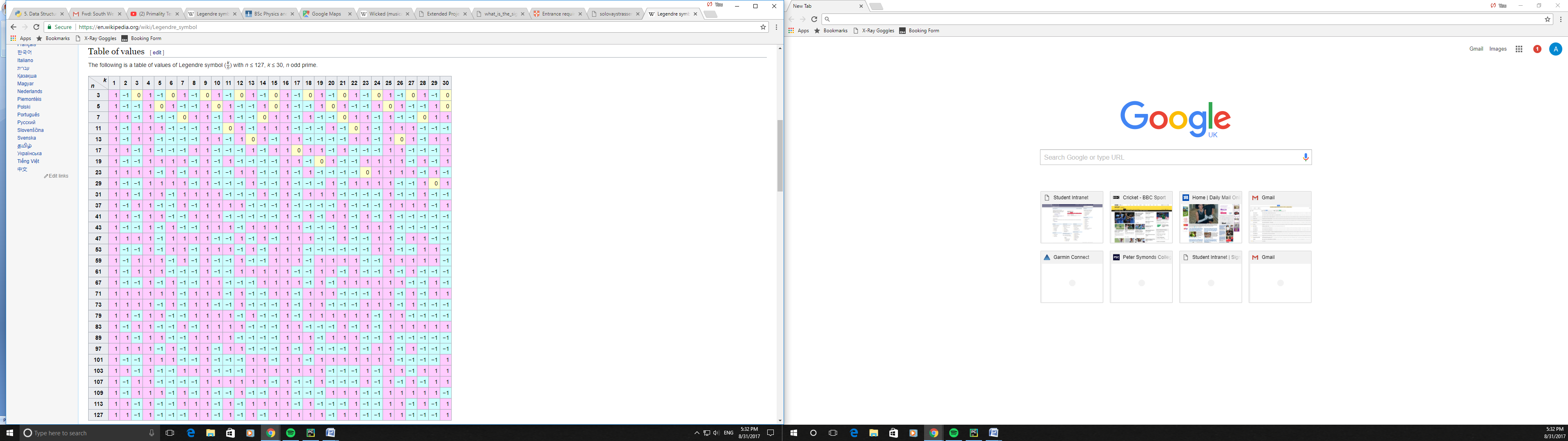


Figure 3- Shows the table of values for the legendre symbol. [23]

1 if is a quadratic residue and

-1 if is a quadratic non-residue modulo

0 if

Quadratic residue modulo means that it is congruent to a perfect square modulo; e.g if there exists integer s.t. .

The algorithm uses these facts in order to run a primality test. Here is an example of the algorithm running [24]:

*If we wish to check whether n = 221 is prime.*

*Selecting a value for that is less than n, say .*

*Using the facts written above this can be rewritten as:*

*The results from this first test inform us that either 221 is prime or 47 is an ‘euler liar’. In this example there is obvious grounds to do another test with a different value of a- 221 has a prime factorisation of 13 \* 17. This time using a = 2:*

*This shows that the congruence does not hold when a = 2, therefore using this method 221 must be composite. This is because the method needs the congruence to hold for all values of a in order to confirm that n is indeed a prime number.*

**Miller-Rabin Primality Test**

The Miller-Rabin primality test is a probabilistic algorithm for deciding whether an odd number is a prime number, with the obvious exclusion of 2. The concept of the algorithm is that a prime will pass the test for all values of a, where a lies in the range 1 a n-1, in actual fact this is proven for 75% of cases up to n-1[25]. For any given odd integer, n, means that n-1 is even and we can write it as where e and d are integers. Using fermat’s little theorem the polynomial , this can then be factorised as many times as there are powers of two in the exponent. This factorisation happens in the form:

Using the factorisation above a can be substituted in (using Fermat’s Little Theorem : ).

When n is a prime number one of these factors will be equal to 0 mod n. S.t.

\*

If n is not prime the congruences still make sense however they may be false for many values of a in the range . \*If these two congruences are false for all then a is a witness for n. But if one is true and another false then a is a non-witness for n (n is often called a strong pseudoprime to base a).[26]

The method has three basic steps to it[28]:

1. The method only works for odd numbers, so exclude two before starting. The first step is factorising in the form .
2. Then one must pick a value that lies in the range 1 < a < n-1.
3. Then finally evaluating whether n is prime by computing , if is equal to either 1 or -1 then n is most probably prime. If equal to neither, one must compute , keep computing until is equal to either 1 or -1. If is equal to 1 then n is composite but if n is equal to -1, n is most probably prime.

An example of the algorithm[28]:

*If n = 53, we can write n-1 (in form ) as . Then a value for a must be picked in the range , say 2.*

*Does No it comes out as 30, therefore we need to calculate .*

*Does Yes it comes to -1, therefore it is most probably prime.*

**Section 3**

**3.1-Discussion: Parameters** **of investigation**

This investigation aims to provide data in order to compare with both data and limitations of the methods in question from my research. The investigation will be all written in python, specifically python 3, in order to keep continuity between methods and will be in the IDE (Integrated Development Environment), Pycharm. The different methods will all be implemented for given limits, of varying magnitudes, in order to test the speed and accuracy of the value given for . I have not implemented the Solovay Strassen algorithm and the Riemann Explicit formula but information upon the run time and efficiency of the algorithm will be used in evaluating all of the algorithms.

The algorithms will all be run for a series of predefined values of . These values of will be placed into a table format in order to compare and contrast various aspects of their run effiency.

**3.2-Ease of implementation**

**Legendre’s Constant**

The method is incredibly simple to implement, with the code occupying only a handful of lines. It however was proven to give what essentially could be called an unusable answer since the constant was proven to be 1 rather than 1.08366, if there is a constant at all. This perhaps is the price that you pay for having such an easy method to implement.

**Methods of Gauss**

The first method of Gauss, designed when he was only 15 years old, is virtually the same as the Legendre method but with the absence of a constant. The method is equally as simple to write and produces very similar results. The implementation of both the first method of Gauss and Legendre’s constant requires the *math library* in python in order to take the natural log of

However, the function in its purest form:

This, the logarithmic integral function, proves particularly difficult to solve using pre-university maths. The implementation time can be greatly reduced by using the shorter series for which the first three terms produce a better estimate than the first method of Gauss.

This is the method of approximating , as both sides are asymptotic to each other, that I have used to implement the method. It is relatively simple to implement using a for loop and a given value for infinity. This value of infinity can greatly change the size of the resultant value for

**Riemann Hypothesis**

Van Helge Koch derived an error term for that is logically equivalent to Riemann Hypothesis, given that the Riemann Hypothesis is true.

This becomes a useful addition to the use of the series for as it enables the correction of the function for values . The implementation is not difficult as for values as for values below this limit are always greater than the value of , for this project I have made the assumption that will always be below this value. This made the implementation of this error term particularly simple as no condition was required in order to change the sign when . Thus the error term was useful addition to the function.

The implementation of the Riemann’s explicit formula is a task of enormous proportions. It requires deep understanding of both the mathematical side of complex numbers as well as the techniques required for implementing these in code. Not only is it necessary to implement R(), a function itself that requires a summation of over a value from the zeta function, but also the summation to infinity of R(). And before you can calculate the value of R() you must already have a formula or at worst, a list of the non-trivial zeros of the zeta function. This means that the implementation of the Riemann’s formula is an entanglement of complex methods. Unfortunately due to these issues an implementation of the formula has not been produced for this project but secondary data has been found in order to form an evaluation.

**Sieve of Eratosthenes**

The sieve requires only the most basic operators, only using addition and multiplication in the version implemented. This makes it both a simple and time efficient method to implement. The method is very easy to prove its success, as it checks whether it removes every prime multiplied by a whole number up to the given limit. For instances it removes in its first iteration: 2\*1, 2\*2,..., 2\*n.

There are many alterations that can be made in order to make it more efficient that can change the implementation and therefore how complicated it is to implement. But for the purposes of this project the simplest version of the algorithm has been implemented.

**Trial Division**

The trial division algorithm is the first of the algorithms to use the modulus function in order to indicate the remainder of a division. In many respects it is very similar to the Sieve of Eratosthenes but the Sieve removes every multiple of an integer whereas Trial Division takes an integer and tries to divide every integer less than itself. Both algorithms have a very similar ease of implementation, for that reason making them a logical solution with respect to complication.

Alternatives, likewise with the Sieve, can produce much more efficient results. An instance of an improvement to the Trial division algorithm would be that you needn’t go further than of a number as it will have already been picked up. This greatly reduces the amount of division trials that are required for the algorithm.

**Fermat Primality Test (Fermat’s Little Theorem)**

The test is probabilistic meaning it can only determine whether a number is probable to be prime. It relies up a simple congruence which uses modular arithmetic. The algorithm randomly selects a value for (see Pg X), this then uses the congruence suggested by Fermat’s Little Theorem for every value up until the limit that is selected- using a For loop. One addition was made to the algorithm, that was to add a section that checked another value of in order to check for Fermat liar and ensure that the original value for really is a witness.

The algorithm was all-round one of the more difficult methods to implement. This is the case for several reasons, firstly, the algorithm requires a significant amount of tweeking (involving loops and functions) to make it count primes up to a given limit. Secondly, the original algorithm is crippled by issues such as Carmichael Numbers (mentioned pgX) which mean that extra measures have to be put in place in order to check for errors. One of these extra measures was the second check for the value of . These factors meant that a more complicated algorithm had to be implemented, thus an algorithm that was no easy to implement.

**Miller-Rabin Primality Test**

The Miller-Rabin algorithm most definitely holds the crown for the most time taken to implement. The different ‘paths’, dependent upon the previous outcome, make it a complicated algorithm to program. The many ‘paths’ ensure that many different conditional statements are required in addition to a large proportion of the code being with functions and loops.

The algorithm already becomes so complicated when you apply it to the case of the amount of primes less than a given quantity that no extra additions were added to this test. However significant alterations can massively perplex one whilst also improving the outcome. For this reason the algorithm has to be the most difficult to implement really well, with the standard algorithm producing very modest results but much better results can be found with increased refinement.

**Conclusion to ease implementation**

Clearly the simplest methods are far easier to implement than others. The more mathematical methods e.g. Gauss and Li(x), are far less complicated than the ‘algorithmic’ methods and for this reason are much easier to implement. But ease of implementation tends, not in all cases, to give worse results than more complex algorithms. For instance the first method of Gauss is definitely one of, if not, the most elementary implementations. However, this method does give poor percentage errors that decrease very slowly as values of increase, 6.4% for when according to appendix A. When placed in the bigger picture this percentage difference becomes particular poor compared to a more sophisticated algorithm such as the Sieve of Eratosthenes which has a percentage error of 0% for all values of .

**3.3-Results from implementation**

The results from the implementation are showcased in Appendices A through I. The table in Appendix A compares the speed (time), outcome and percentage error for all of the methods. The best comparison is to compare results from runs using larger numbers ().

**Legendre Constant**

The method produced results with considerably high percentage errors (10%+) for values less than 1000. But for larger values the percentage errors appear to be getting much smaller, with less than 0.1% for . Perhaps the Legendre isn’t such a rotten method as it is portrayed to be, since the percentage errors are considerably lower, in general, than other algorithms for instance the Miller-Rabin test (31.4% for ).

**Gauss’s First Method**

This method really does illustrate the case that the methods that are easier to implement do give much worse percentage errors. Although the percentage error does slowly decrease, it decreases at such a slow rate that it appears that it takes too long to be a particularly useful method (reaching 6.64% at when ).

**Gauss’s Second Method-**

The function appears to decrease in percentage error at a considerably faster rate than either of the two methods mentioned before. For instance the percentage error is less than 1% by the time = and is at 0.00013% error by the time . This appears to confirm other sources in saying that , as the percentage error between the two seems to be converging to the value of 1.

**with Koch Error term**

The data collected from the implementations seems to suggest that the rate at which the function with error term is much slower than just the function. This seems slightly counterintuitive however; Van Helge Koch stated that the error term was only applicable when . This may explain why the rate is slower considering the data is only using a small range of values for . Also the intervals between each successive value of the error term are getting smaller at a faster rate than for once .

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Gauss | Time | Error % |  | Time | Error % | Legendre | Time | Error% |
|  | 4 |  |  |  |  |  |  |  |  |  |
|  | 25 |  |  |  |  |  |  |  |  |  |
|  | 168 |  |  |  |  |  |  |  |  |  |
|  | 1229 |  |  |  |  |  |  |  |  |  |
|  | 9592 |  |  |  |  |  |  |  |  |  |
|  | 78498 |  |  |  |  |  |  |  |  |  |
|  | 664579 |  |  |  |  |  |  |  |  |  |

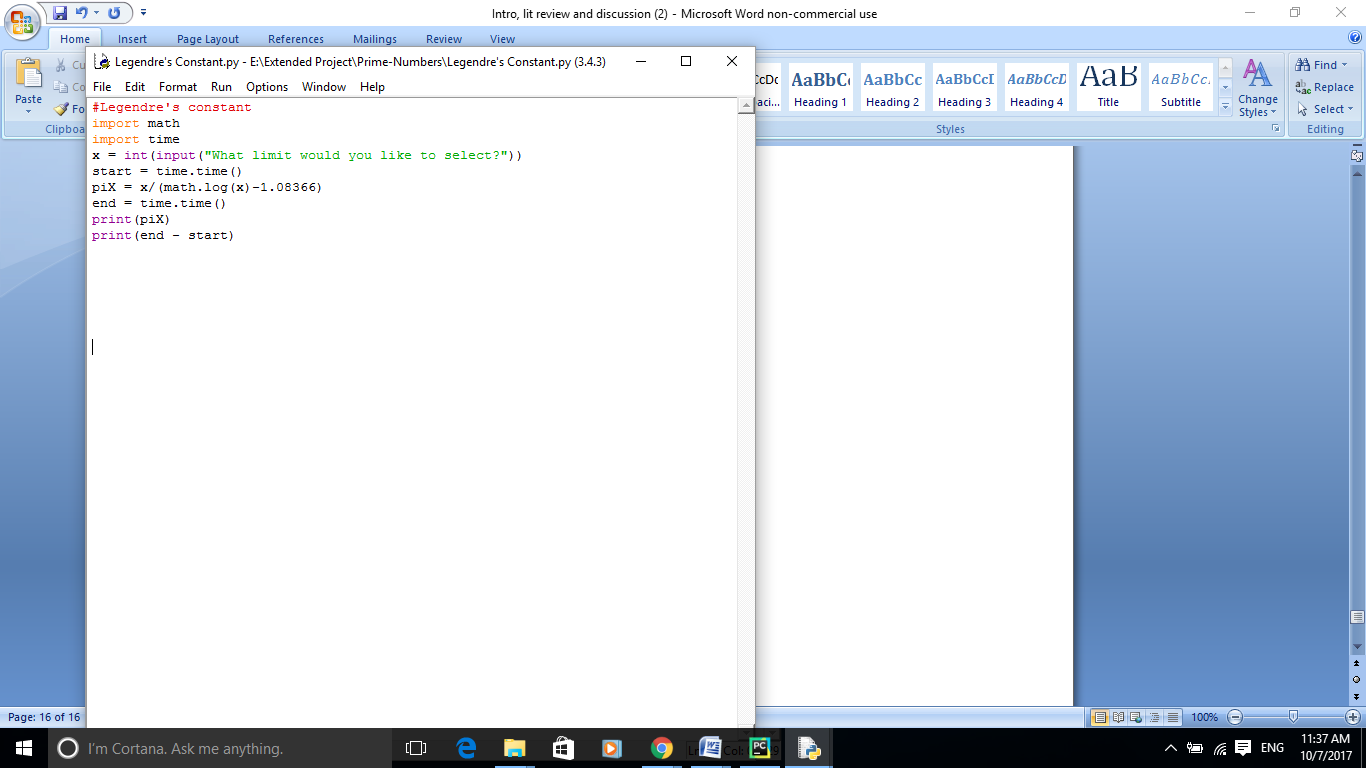
**Appendix A-Results from Implementation**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Gauss | Time(s) | Error % |  | Time(s) | Error % | Legendre | Time(s) | Error% |
|  | 4 | 4.3429448 | 0.0000000 | 8.57362 | 10.0017854 | 0.0000000 | 150.045 | 8.2039496 | 0.0000000 | 105.099 |
|  | 25 | 21.714724 | 0.0000000 | 13.1411 | 29.8118813 | 0.0000000 | 19.2475 | 28.396907 | 0.0015430 | 13.5876 |
|  | 168 | 144.76483 | 0.0000000 | 13.8305 | 174.424457 | 0.0000000 | 3.82408 | 171.70049 | 0.0000000 | 2.20267 |
|  | 1229 | 1085.7362 | 0.0000000 | 11.6569 | 1237.55410 | 0.0000000 | 0.69602 | 1230.5147 | 0.0000000 | 0.12325 |
|  | 9592 | 8685.8896 | 0.0000000 | 9.44652 | 9605.54872 | 0.0000000 | 0.14125 | 9588.4030 | 0.0000000 | 0.03750 |
|  | 78498 | 72382.414 | 0.0000000 | 7.79075 | 78544.7789 | 0.0000000 | 0.05959 | 78543.178 | 0.0000000 | 0.05755 |
|  | 664579 | 620421.69 | 0.0000000 | 6.64440 | 664578.123 | 0.0009999 | 0.00013 | 665139.70 | 0.0000000 | 0.08437 |
|  |  | Li(x) ET | Time(s) | Error % | FLT | Time(s) | Error % | M-R | Time(s) | Error (%) |
|  | 4 | 9.7120667 | 0.0000000 | 143.552 | 3 | 0.0000000 | 25 | 4 | 0.0005009 | 0 |
|  | 25 | 27.979542 | 0.0000000 | 11.9182 | 24 | 0.0000000 | 4 | 25 | 0.0030019 | 0 |
|  | 168 | 165.73291 | 0.0000000 | 1.34946 | 176 | 0.0269999 | 4.76190 | 144 | 0.0420280 | 14.2857 |
|  | 1229 | 1200.9073 | 0.0000000 | 2.28582 | 1280 | 23.467000 | 4.14972 | 868 | 5.2316420 | 29.3735 |
|  | 9592 | 9460.6895 | 0.0000000 | 1.36896 |  |  |  | 6578 | 2838.9100 | 31.4 |
|  | 78498 | 77995.076 | 0.0000000 | 0.64068 |  |  |  |  |  |  |
|  | 664579 | 662550.10 | 0.0000000 | 0.30529 |  |  |  |  |  |  |
|  |  | Sieve | Time(s) | Error % | Trial Div | Time(s) | Error % | Solovay Str | Time(s) | Error (%) |
|  | 4 | 4 | 0.0000000 | 0 | 4 | 0 | 0 | - | - | - |
|  | 25 | 25 | 0.0130091 | 0 | 25 | 0.0020001 | 0 | - | - | - |
|  | 168 | 168 | 6.2143230 | 0 | 168 | 0.1160800 | 0 | - | - | - |
|  | 1229 | 1229 | 4634.9020 | 0 | 1229 | 10.896294 | 0 | - | - | - |
|  | 9592 | - | 6+ days |  | 9592 | 1107.9213 | 0 | - | - | - |
|  | 78498 | - | - |  |  |  |  | - | - | - |
|  | 664579 | - | - |  |  |  |  | - | - | - |

https://primes.utm.edu/howmany.html

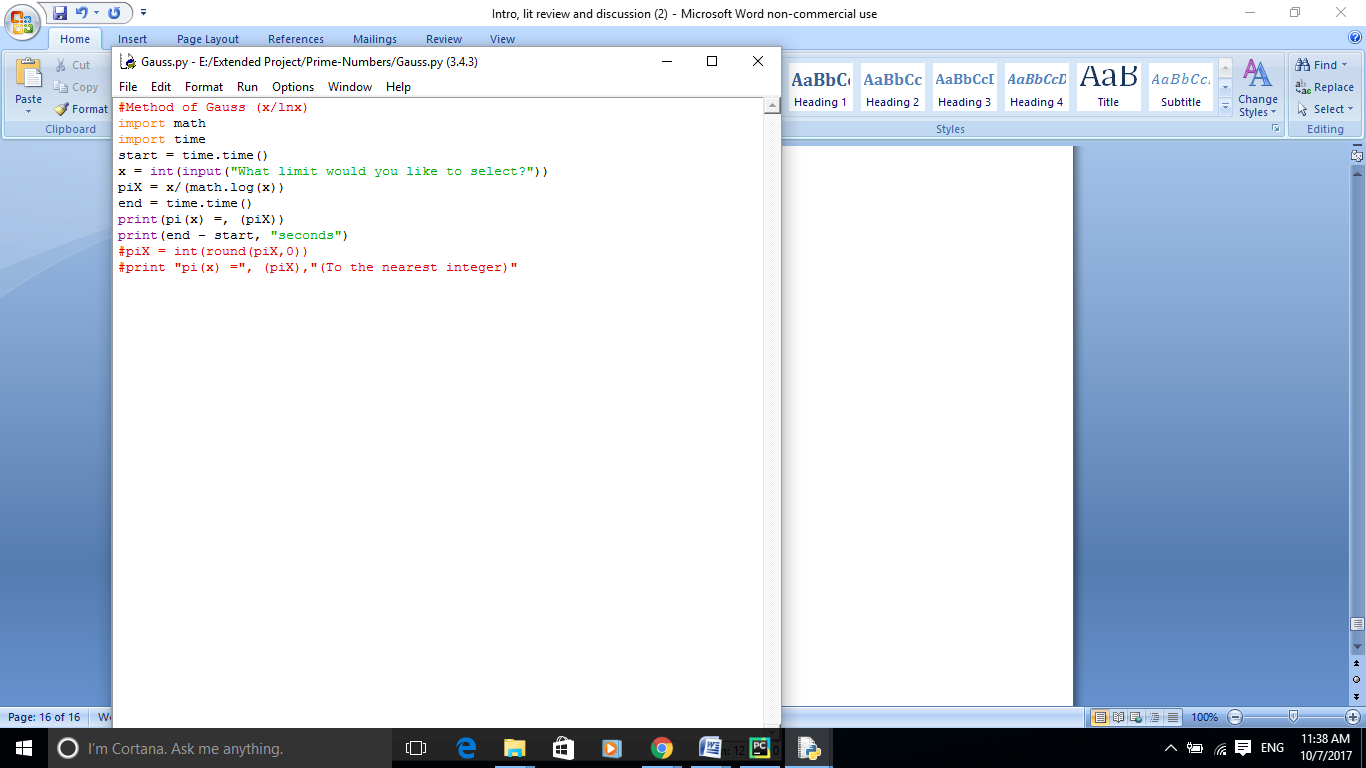
**Appendix B**

**Implementation of Legendre Constant**

****

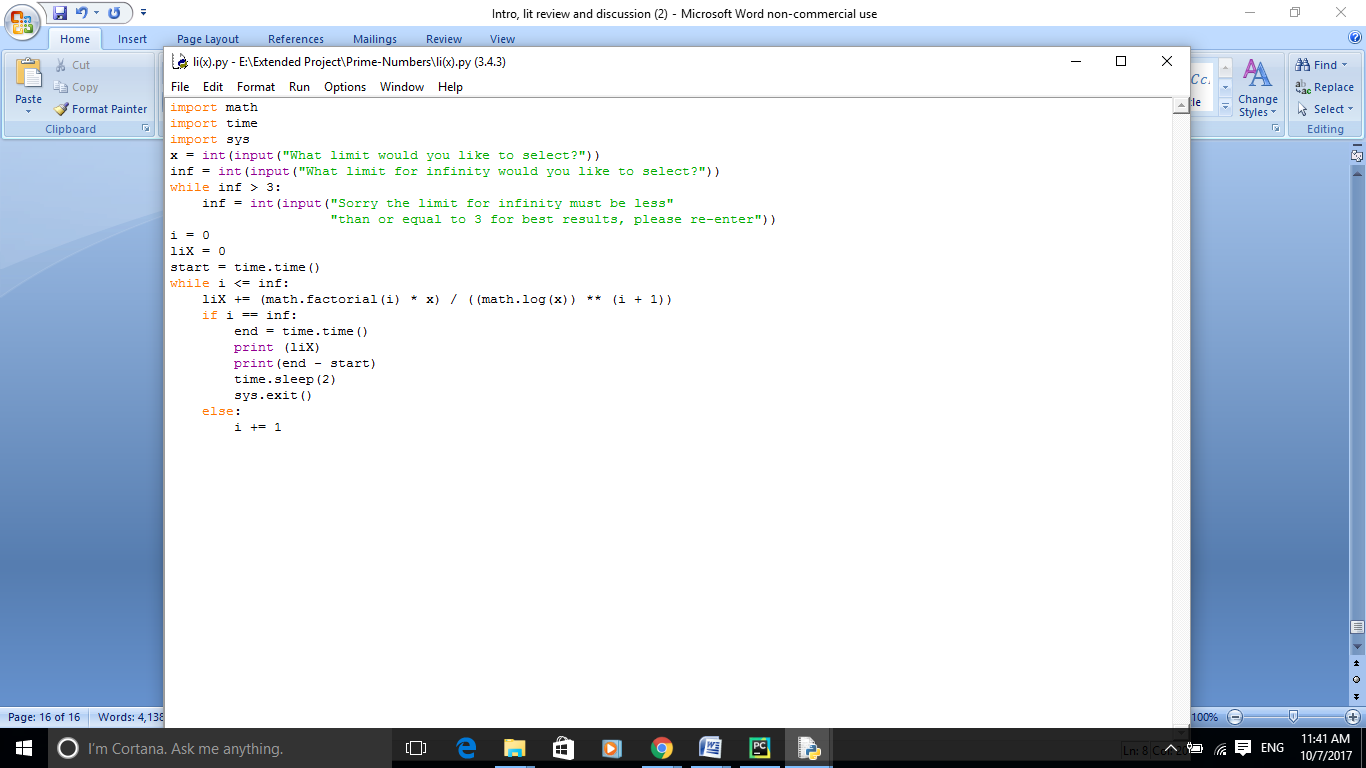
**Appendix C**

**Implementation of Gauss’s method**



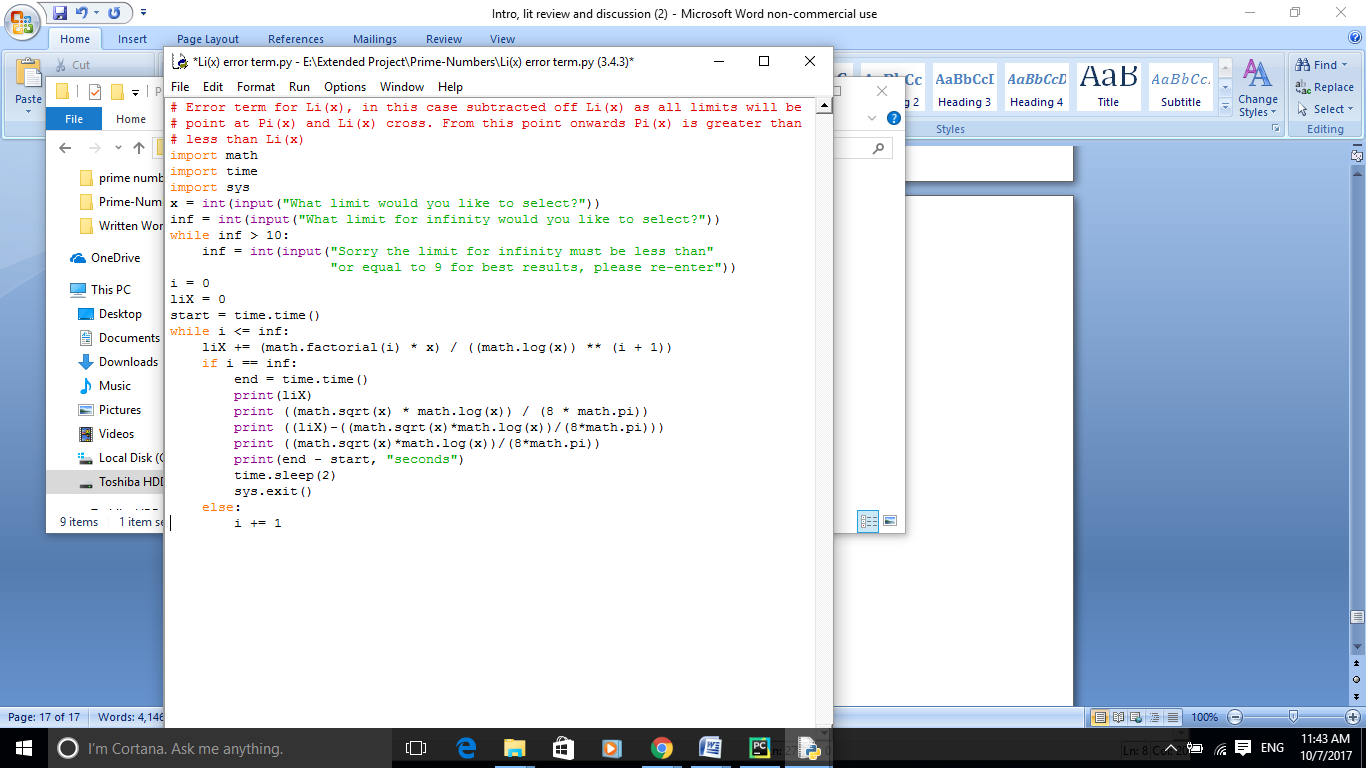
**Appendix D**

**Implementation of**

****

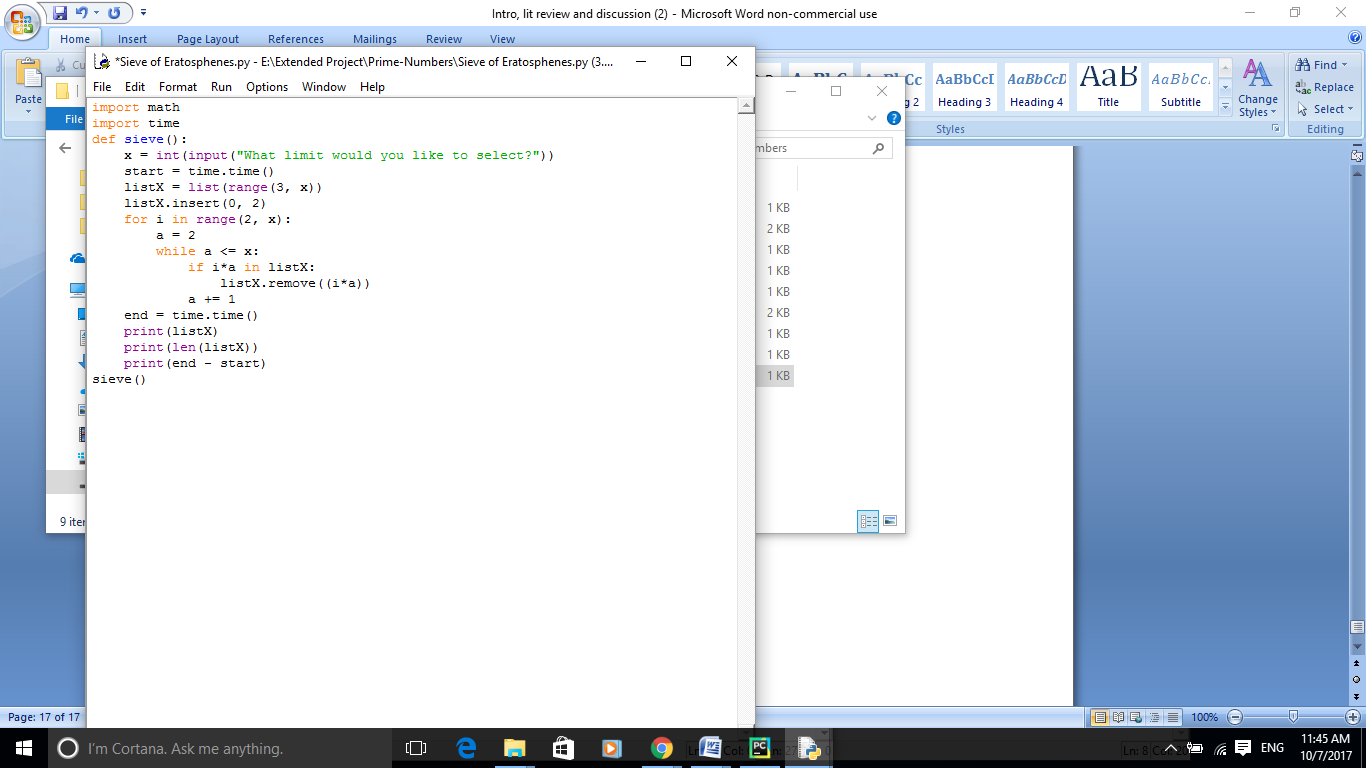
**Appendix E**

**Implementation of the error term**



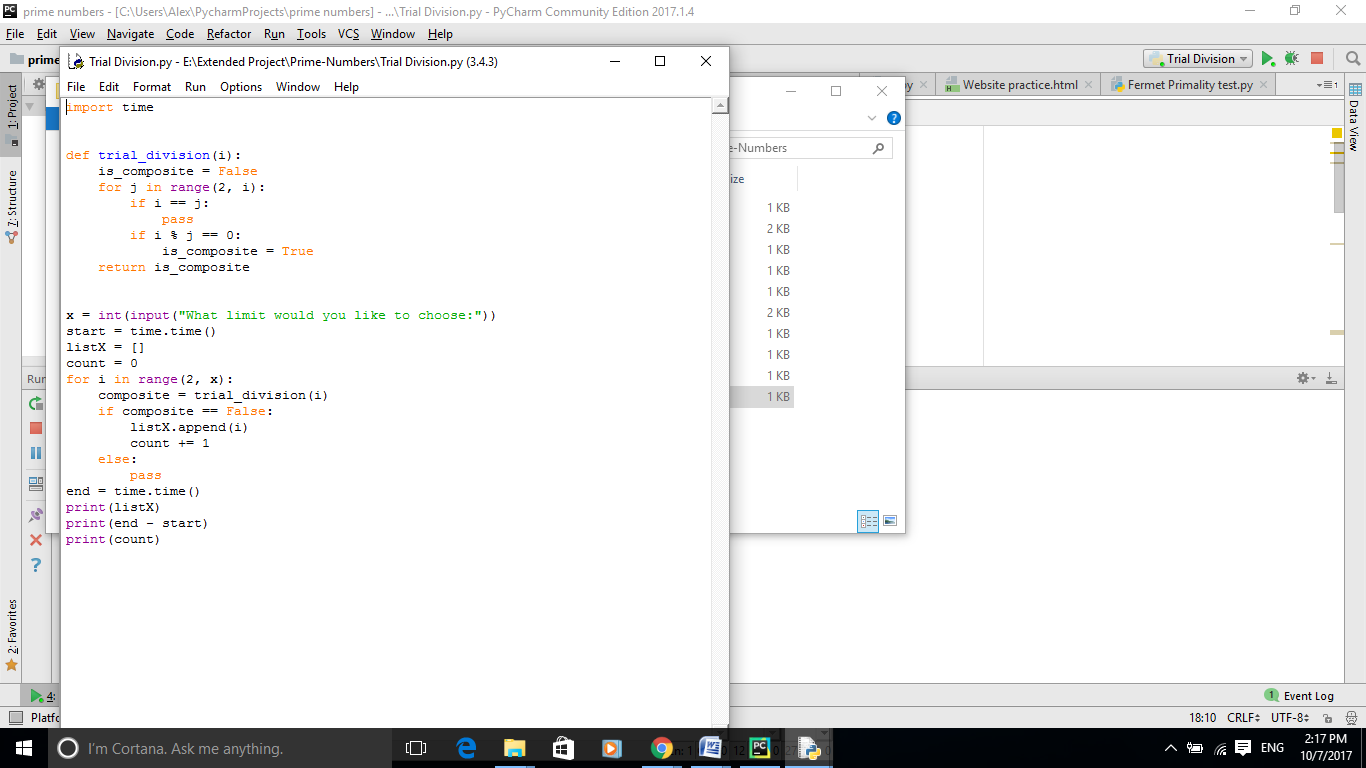
**Appendix F**

**Implementation of Sieve of Eratosthenes**

**

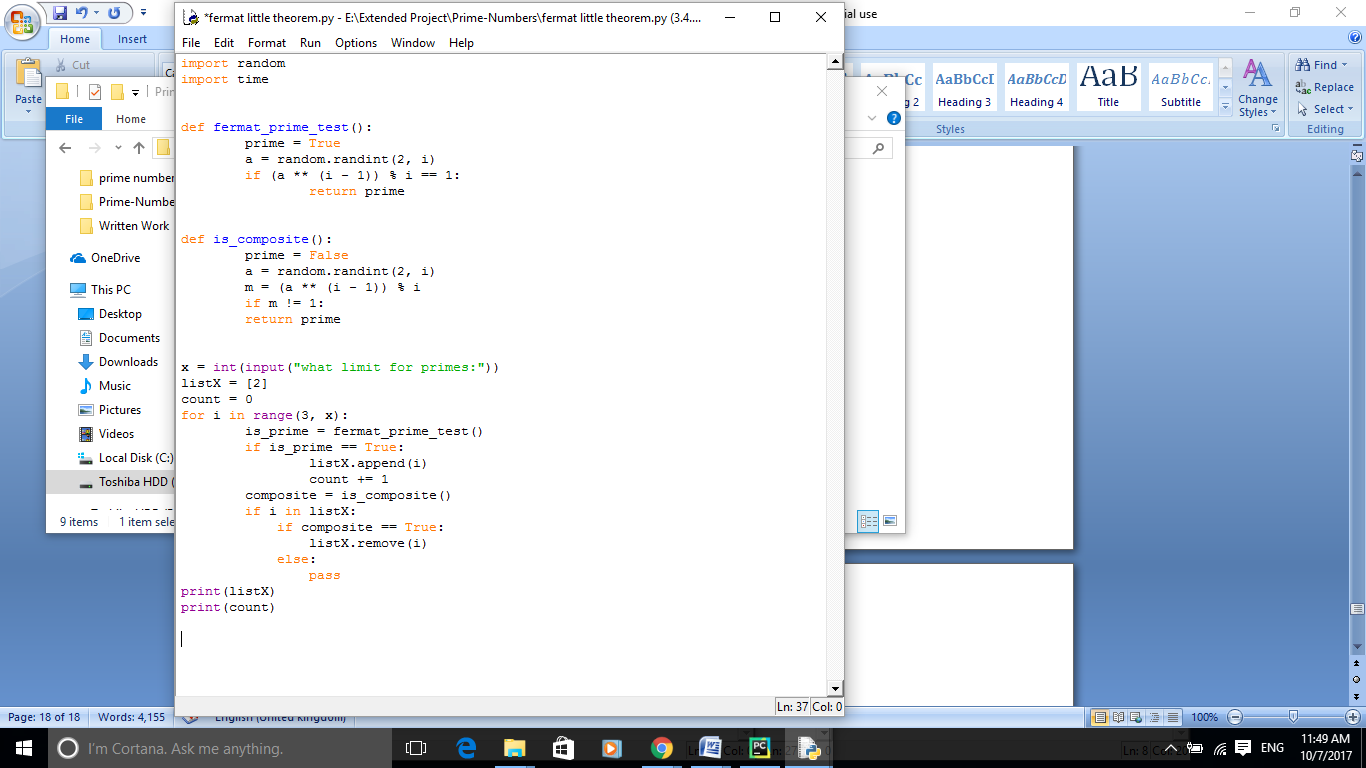
**Appendix G**

**Implementation of Trial Division**

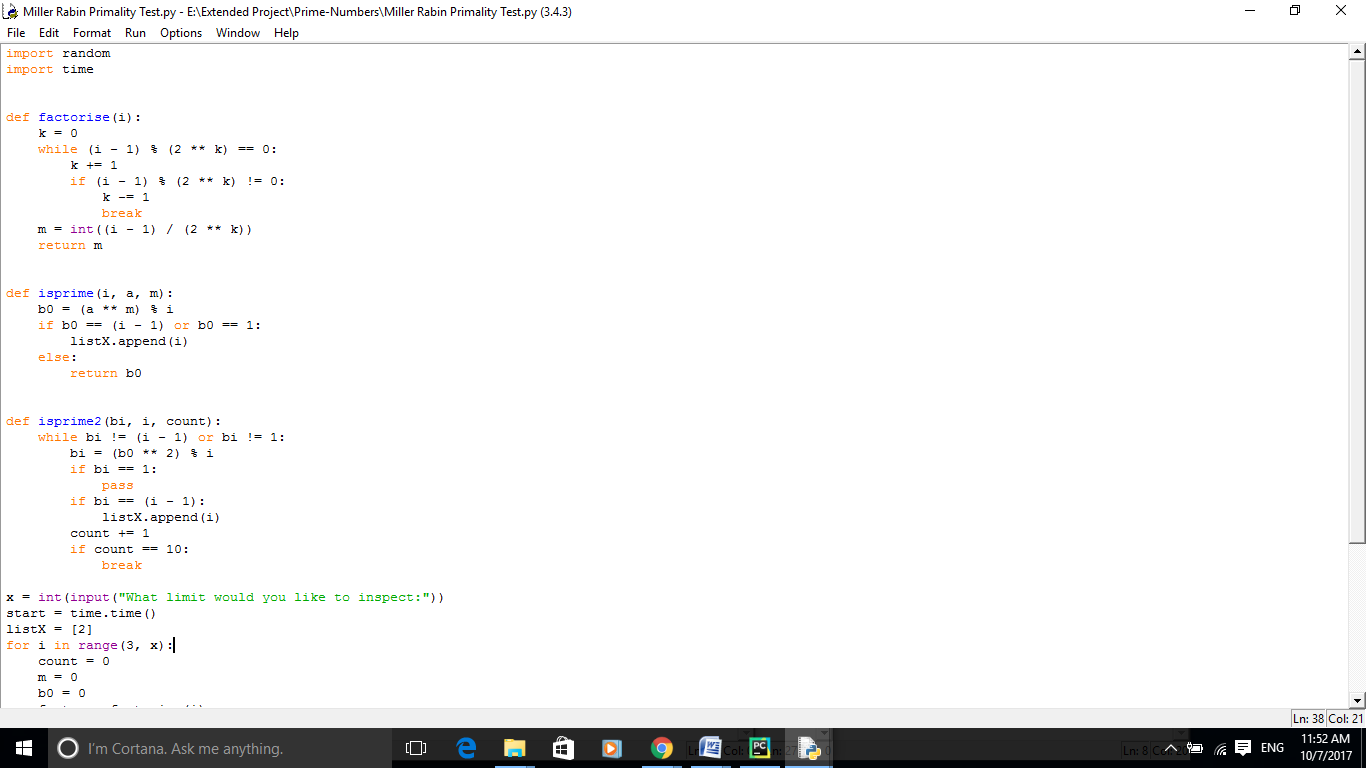
****

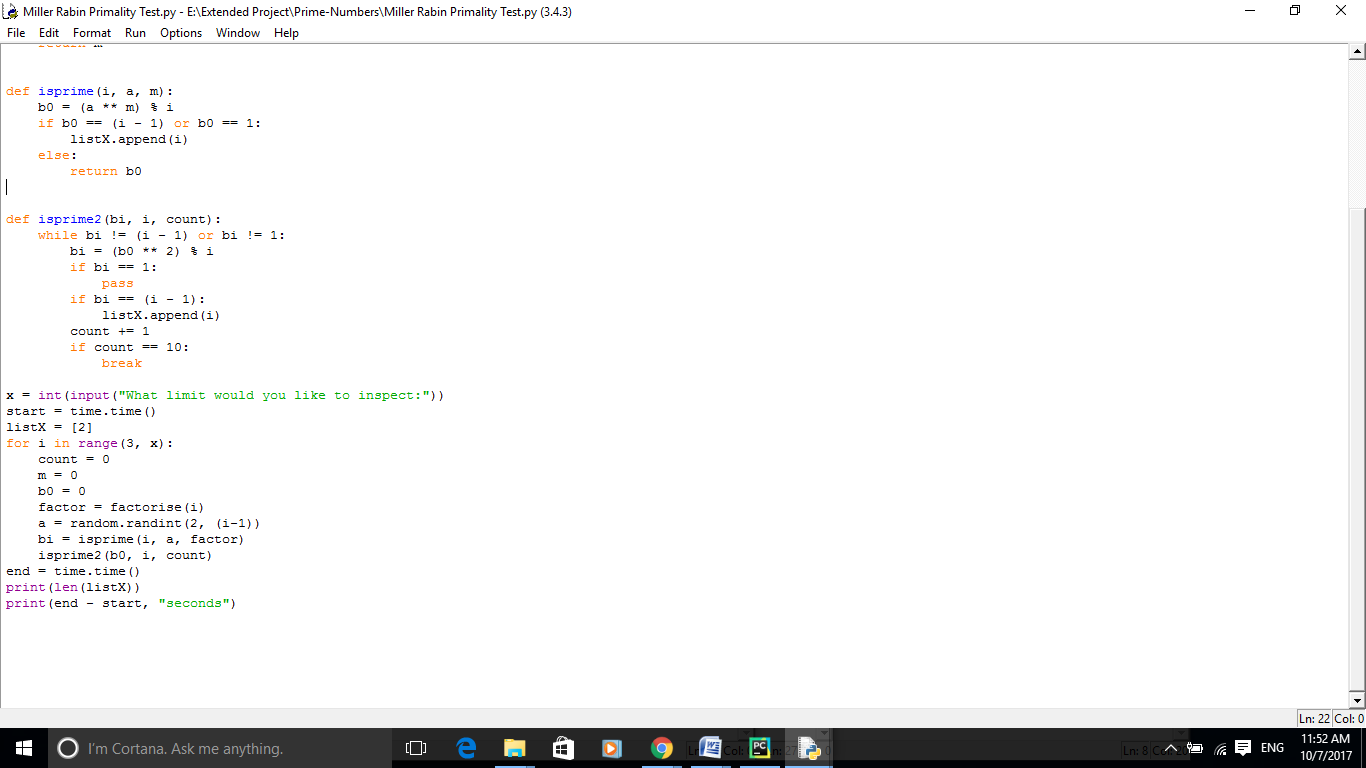
**Appendix H**

**Implementation of Fermat’s Primality Test**

****

**Appendix I**

**Implementation of Miller-Rabin Primality Test**

****

**Appendix J- Repeat Table of Riemann Explicit formula**

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Overestimate | Difference (%) |
|  |  | 1 | 25 |
|  | 26 | 1 | 4 |
|  | 168 | 0 | 0.0000000 |
|  | 1227 | -2 | 0.1627339 |
|  | 9587 | -5 | 0.0521268 |
|  | 78527 | 29 | 0.0369436 |
|  | 664667 | 88 | 0.0132415 |
|  | 5761552 | 97 | 0.0016836 |
|  | 50847455 | -79 | 0.0001554 |
|  | 455050683 | -1828 | 0.0004017 |
|  | 4118052495 | -2318 | 0.0000562 |
|  | 37607910542 | -1476 | 0.0000039 |