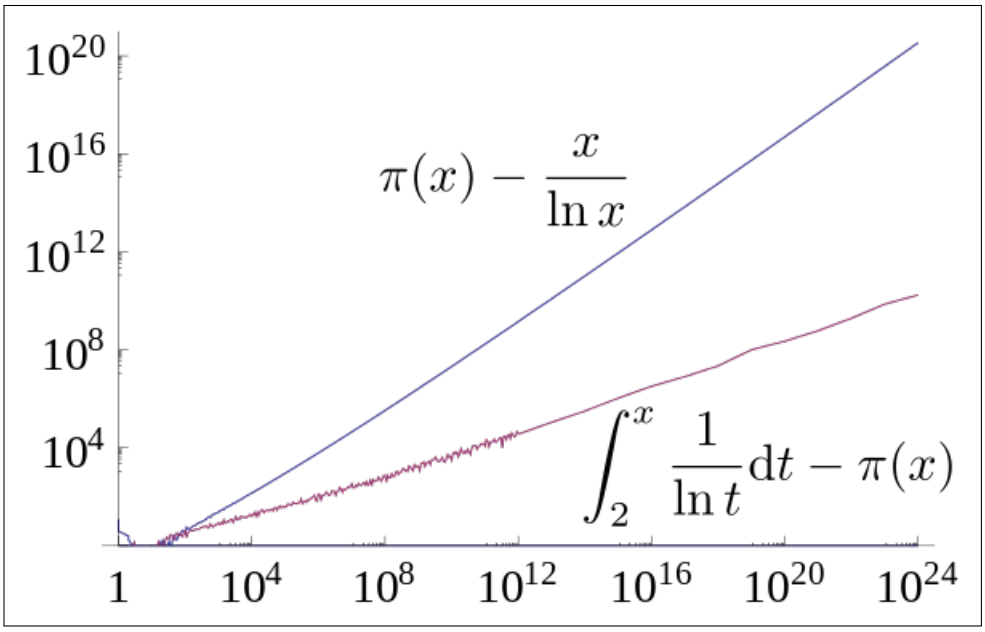
Halmstad
KATTEGATTGYMNASIET

Prime Numbers

Understanding How the Largest Primes Came to Be

Ibrahim Abdulhussein,
Robin Brusbo

30 April, 2020



1

[1] The Prime Number Theorem [14]

Abstract

Prime numbers are a fundamental part of number theory. Primes have many special properties that can be exploited. One of the most common applications being RSA-encryption, which uses a product of two large primes as a key. The rise of more powerful computers has led to weaker keys being cracked. Therefore it is in a Computer Scientists' interest to find larger primes. In order to accomplish this, efficient algorithms have to be developed. This study investigates the efficiency of five simple primality testing algorithms:

- Brute-force
- Lucas-Lehmer Test
- Smart Brute-force
- Fermat Primality
- Miller-Rabin

In the first test, the algorithms were given the task of finding the largest prime number within 60 seconds. The test is performed by looping through the natural numbers starting from 2. The first test could not include Lucas-Lehmer, as it only works for mersenne numbers. In the first test it was determined that the Fermat primality test found the largest prime, followed by Lucas-Lehmer.

In the second test, the algorithms were given mersenne primes of different sizes and returned the time it took for them to finish verifying that it was a prime. However, if the time exceeded 120 seconds, the test case would be deemed as a "failure" and immediately end the program. These tests concluded that Lucas-Lehmer was the fastest algorithm to verify mersenne primes.

The purpose of this study is not to find newer primes, but rather to demonstrate the correlation between the Big O notation and runtimes, as-well as giving an introduction to primality testing algorithms.

Contents

1 Introduction

Prime numbers have many uses in everyday life, although not noticeable. The most common example is RSA-encryption [15], where a product of two *extremely* large primes are used as a key for its toughness to factor. If one of the factors were to be known, obtaining the other factor would be no challenge. If no factors are known, then the only solution is to factor the prime for hand, which is practically impossible.

In the age of computers, keys that used to be considered impossible are now easily cracked. With the rise of more and more powerful computers, these factors are becoming easier to find, thus breaking the weaker RSA-encryption. Therefore it is essential to find larger primes.

Finding larger primes is however far from trivial. Since the largest prime known to man has $24,862,048$ digits [11], using a method that checks all numbers below n is simply not fast enough for large primes.

This study will focus on five algorithms for finding primes, and compare them. The following questions arise:

1. What are some algorithms for primality testing?
2. Which algorithm is the most efficient and why?
3. How far away from the current largest prime can we get?

2 Theory

2.1 What Is a Prime?

Prime numbers are defined as *positive integers* which only have the factors 1 and itself. Thus 4 is not a prime since $4 = 2 * 2$. On the other hand 5 is a prime since the only divisors of 5 is 1 and 5. If the number, n, is not prime, it is referred to as a *composite number*.

An exception to this definition is 1, since it is the first natural number. Subsection (2.3) provides a more in-depth definition of prime numbers, and a mathematical explanation about why 1 is not a prime number.

2.2 Mersenne Primes

A mersenne prime is a prime that can be written as $2^p - 1$ where p is also a prime number. An example of this is $2^5 - 1$ which equals to the prime number 127. The eight largest prime numbers are mersenne primes [11]. The reason being because of their easily exploitable properties by computers, making them faster to test for primality than normal integers.

2.3 The Fundamental Theory of Arithmetic

The Fundamental Theory of Arithmetic [10] states that all integers greater than 1 is either a prime, or can be expressed as a product of primes in a unique way. This means that all natural numbers, except for 1, has its own factorisation containing only primes, unless it is a prime itself.

This study relies on the fact that there is an infinite amount of primes. The proof for this is a simple proof by contradiction:

Assume that there is a finite amount of primes and make a list of them:

$$p_1, p_2, p_3, p_4, p_5, \cdots$$

Let the constant Q be the product of all the primes in the list plus 1:

$$Q = p_1 * p_2 * p_3 * ... + 1$$

According to the fundamental theorem of arithmetic, Q must be a prime since none of the primes in the list divide Q evenly because of the 1; therefore making the list incomplete and proving that you cannot make a finite list of all primes.

2.4 The Prime Number Theorem

The Prime Number Theorem [14] describes approximately how many primes there are less than or equal to a given number. The function $\pi(N) \sim \frac{N}{\ln(N)}$ gives the expected amount of primes below a certain N. Graphing this function shows that primes become less common for greater N.

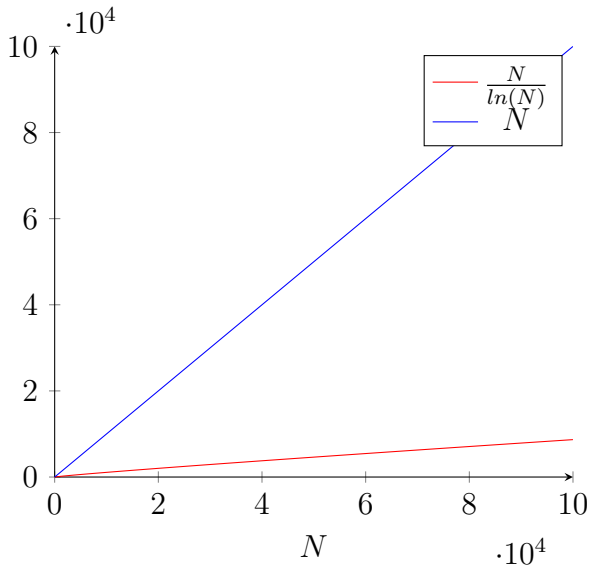


Figure 1: The graph of $\pi(N)$ and N from 0 to 10^5 can be used to compare the relationship between the number N and the approximate amount of primes below it.

Worth to note is that $\pi(N)$ looks linear in this graph. The function is not actually linear, but because of the graph's large domain, the functions seems like it is linear.
This shows that primes do not show up linearly, meaning a computer that is twice as powerful will *not* produce twice as many primes. Instead, the largest factor for verifying primes faster is the *algorithms*.

2.5 Time Complexity

Time complexity [16] is a concept within computer science, which describes the approximate time for a program to complete. The study will use of the Big O Notation [5], which notates how the run time increases as the input size increases. For example, $\mathcal{O}(N)$ will grow linearly with the input size. Increasing the input size by a factor of 10, will also increase the run time by a factor of 10, as such $\mathcal{O}(10N)$. On the other hand, $\mathcal{O}(\log(N))$ grows logarithmically, which is far more efficient for bigger input sizes, as $\mathcal{O}(\log(N))$ is strictly smaller than N for large enough values. The base for logarithms in the Big O Notation is not relevant. The proof as to why the base is irrelevant will not be provided.

The amount of operations a modern computer can do is in the order of magnitude of 10^9 per second. An operation is, for example, adding two numbers or storing a number in an array.

Since the Big O Notation denotes the growth of the runtime as $\lim_{N\to\infty}$, two notations can be added together rather easily. For example, a program that has two bits of codes, one with $\mathcal{O}(N)$ and another with $\mathcal{O}(N^2)$, will have an overall time complexity of $\mathcal{O}(N^2)$ since N^2 dominates for large values of N.

2.6 Deterministic vs. Probabilistic

A deterministic test is a one-hundred percent definite test for primes, that gives no false positives. It either returns true if the number being tested if

prime, or false if the number is not prime.

Probabilistic tests differ from deterministic tests by having a stochastic, random factor in them. They are not one-hundred percent definite and therefore sometimes will give false positives/negatives.

An example of a probabilistic test for the number p would be to test a random number below p and see if it divides p evenly. This test would give a lot of false positives, because even if p were to be composite, not all numbers below it will divide it. The accuracy of the test, i.e. the probability that it returns a correct answer, is higher if there is an increased amount of unique factors, k, tested.

2.7 Algorithms

2.7.1 Brute-force

The brute force method is a deterministic method that tests all the integers, n, between 2 and $p-1$ and checks if n satisfies $n \equiv 0 \pmod{p}$. If the condition is satisfied, n would be a divisor of p which would make p not a prime. However, if the loop completes without finding any divisors to p, then p is prime. The brute force algorithm is a definite test, since it follows the definition of a prime; p is prime if and only if there are no divisors except for 1 and p itself. The algorithm's time complexity is $\mathcal{O}(N)$, meaning it increases linearly. The code for this algorithm is shown below. All code in this study will be written in Python. The reason behind this will be discussed later.

```
def brute(n):
    for x in range(2, n): # Loops x from 2 to n-1
        if n % x == 0: # If x divides n evenly
            return False
    return True
```

2.7.2 Smart Brute-force

The smart brute force is a variant of the brute force algorithm (2.7.1). By utilising some properties of primes, the smart brute force has a time complexity of $\mathcal{O}(\sqrt{N})$ compared to the $\mathcal{O}(N)$ complexity from the brute force. Just like the original brute-force, this algorithm is deterministic aswell.

Since a prime can never have the factor 2 in them, it is sufficient to only test odd numbers. Assuming that p is a composite, it must have at least 2 factors. Therefore, the smart brute force only tests numbers up to $\sqrt{p}$, since if a factor larger than $\sqrt{p}$ were to be found, the others must have already been tested.

```
import math


def smartBrute(n):
    if n % 2 == 0:
        return False
    if n == 1:
        return False
    if n == 2:
        return True
    for x in range(3, math.isqrt(n) + 1, 2):
        if n % x == 0:
            return False
    return True
```

2.7.3 Lucas-Lehmer

The Lucas-Lehmer algorithm [12] is a deterministic primality test that only works for Mersenne numbers. It takes advantage of the special properties of mersenne numbers:

For some $p > 2$, $M_p = 2^p - 1$ is prime if M_p divides S_{p-2} where $S_0 = 4$ and $S_k = (S_{k-1})^2 - 2$ for $k > 0$.

The Lucas-Lehmer test has helped the GIMPS (Great Internet Mersenne Prime Search) [4] to find many of the largest primes known. This is because the time complexity of the algorithm is much faster compared to the other algorithms. The time complexity being $\mathcal{O}(\log^2(N) \log\log N)$.

```
def lehmer(n):
    s = 4
    M = pow(2, n) - 1 # M = 2^n-1
    for _ in range(n - 2): # loops from 1 to n-2
        s = pow(s, 2, M) - 2 # s = s^2 mod M - 2
    if s == 0:
        return True
    else:
        return False
```

2.7.4 Fermat

The Fermat primality test [8] is a probabilistic algorithm. Its whole concept is based on Fermat's little theory [9] which states:

If p is prime and a is not divisible by p then:
$a^{p-1} \equiv 1 \pmod{p}$

To implement the Fermat test, the code randomises an integer for a and checks if it agrees with the theory. If it does not, then p cannot be prime. However if it does, then it is *probably* prime. The accuracy of the test is higher if it tries more than one a. In the code the amount of *rounds* is determined with the variable k. The time complexity for the algorithm is $\mathcal{O}(k \log^2 N \log\log N \log\log\log N)$. The speed of the test can be compared to Lucas-Lehmer.

```python
import random


def fermat(n, k):
    if n < 5:  # n under 5 cannot be verified
        return False
    for _ in range(k): # loops from 1 to k
        a = random.randint(2, n - 2) # a = random integer between 2 and n-2
        if pow(a, n - 1, n) != 1: # if a^(n-1) is not congruent to 1 mod n
            return False
        return True
```

2.7.5 Miller–Rabin

The Miller-Rabin primality test [13], first discovered by Russian mathematician M. M. Artjuhov in 1967, is based on the Fermat test (2.7.4) and has two variants. One that is deterministic, and the other probabilistic.

The deterministic version is conditional, meaning that it relies on an unproven theorem, in this case, the extended Riemann hypothesis [7]. On the other hand, the probabilistic version is unconditional, meaning that it does not depend on an unproven theorem. This makes the probabilistic version a more reliable primality test. If the extended Riemann hypothesis were to be disproved, the deterministic version would fall apart. The probabilistic variant is also faster for bigger n compared to the deterministic version. For these reasons, the probabilistic version is chosen for this study.

The time complexity for the probabilistic version is $\mathcal{O}(k \log^3(N))$, where k is the amount of rounds the algorithm will perform.

```python
import random


```

```
def findIntegers(n):
    counter = 0
    number = n - 1
    while number % 2 == 0:
        counter += 1
        number //= 2
    return (counter, number)


def millerRabin(n, k):
    if n < 5:  # n under 5 cannot be verified
        return False
    r, d = findIntegers(n)
    for _ in range(k):
        a = random.randint(2, n - 2)
        x = pow(a, d, n)
        if x == 1 or x == n - 1:
            continue
        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == 1:
                return False
        if x == n - 1:
            continue
        return False
    return True
```

3 Materials and Methods

3.1 Resources

The resources for gathering information about the different algorithms and other relevant information, will come from the internet. Wikipedia [17] will be used extensively, since the difficulty of understanding the articles on Wikipedia is not too high, nor too low.

The programming language that will be used is Python [2]. Python is an object-oriented programming language that is often compared to pseudocode because of its simplicity and its English-like syntax. Even though Python is considerably slower [1] than other, more low level, languages, such as C++ [6], the simplicity and ease of understanding it outweighs the drawback.

The tests will be performed on a 2014 HP laptop running the Linux operating system. Since the tests will only be a comparison of the different algorithms, it does not really matter which computer the tests are performed on. However all tests have to be conducted on the same computer.

3.2 Methods

The comparison of algorithms will be split up into two different tests, to test different characteristics of the algorithms.

The first test will be called "Finding Largest Prime in 60 Seconds". Here the algorithms will be given 60 seconds to loop through the natural numbers starting from 2. The algorithms will test the number for primality, and if it is shown to be prime, it will be inserted into an array. Once the time runs out, the largest element in the list is recorded as the result for the test. The test will be repeated 3 times to be able to see if there is any variation between the different instances of conducting the test. The following algorithms will be tested: Brute-Force, Smart Brute-Force, Fermat and Miller-Rabin. The reason for why Lucas-Lehmer is not included is that it only works with Mersenne

numbers.

The second test, called "Largest Mersenne Prime Verified in 120 Seconds", will test Mersenne primes for primality. All algorithms are expected to return true, which is why the runtimes are to be noted instead. This is in order to compare the speed and time complexities of the different algorithms. A test case is deemed a "success" if it finishes within 120 seconds. If it exceeds 120 seconds, or if it still running after five minutes, it is deemed a "failure". A test case that succeeds is coloured green, and red for a failure. The numbers used for the test are the following: 19, 31, 61, 107, 607, 1279, 4423, 9689, 11213, 19937, 23209, 44497 and 86243. Note that these numbers are the exponents of the Mersenne prime being tested, and not the actual numbers being tested.

All the algorithms will be compared, since all algorithms are able to test mersenne primes. For the probabilistic algorithms, the constant k will be equal to 10, an arbitrary number to decide the number of rounds.

In order to conduct these tests, a "driver" function will be used to call the algorithms and start a timer. The code for the different algorithms will be translated from pseudocode into Python and can be found in more detail at the study's GitHub page [3].

Driver function for “Finding Largest Prime in 60 Seconds” (4.1):

```
# This driver file finds the largest prime within
# a minute and prints it.
#
# Change line 35 to desired algorithm, i.e. brute(n).
# WARNING! Lucas-lehmer cannot be used.
#
# Make sure fermat and millerRabin are written as
# <function>(n, 10). Where 10 is k, the second parameter

import time

# from FILE import FUNCTION
from brute import brute
from smartbrute import smartBrute
from lucas import lehmer
from fermat import fermat
from milrab import millerRabin


def driver():
    n = 2
    primes = []
    timelimit = time.time() + 60  # 60 seconds
    while time.time() < timelimit:
        if primeTest(n):
            if time.time() < timelimit:
                primes.append(n)
            else:
                break
        n += 1
    print(max(primes))


def primeTest(n):
    if smartBrute(n):  # Algorithm change here
        return True
    return False

```

```
39
40 if __name__ == "__main__":
41     driver()
```

Driver function for “Largest Mersenne Prime Verified in 120 Seconds” (4.2):

```
# This driver file tests if an algorithm can solve
# a mersenne prime within two minutes. If it cannot,
# the driver code will print the mersenne prime it failed
# on with an 'X'. However, if it succeeded, it will
# instead print an 'O' and the time it took.
#
# Change line 58 to desired algorithm, i.e. brute(n).
#
# Make sure fermat and millerRabin are written as
# <function>(n, 10). Where 10 is k, the second parameter

import time

# from FILE import FUNCTION
from brute import brute
from smartbrute import smartBrute
from lucas import lehmer
from fermat import fermat
from milrab import millerRabin


def driver():
    # 13 exponents for mersenne primes
    # Taken from https://en.wikipedia.org/wiki/Mersenne_prime
    exponent = [
        19,
        31,
        61,
        107,
        607,
        1279,
        4423,
        9689,
        11213,
        19937,
        23209,
        44497,
        86243,
```

```
    ]
    for n in exponent:
        start = time.time()
        timelimit = time.time() + 120  # 120 seconds
        primeTest(pow(2, n) - 1)
        if timelimit > time.time():
            print(
                  str(n) + ": 0\nTime: " +
                  str(time.time() - start) + " seconds\n"
                 )
        else:
            print(
                  str(n) + ": X\nTime: " +
                  str(time.time() - start) + " seconds\n"
                 )
            return


def primeTest(n):
    if smartBrute(n):  # Algorithm change here
        return True
    return False


if __name__ == "__main__":
    driver()
```

4 Results

4.1 Finding Largest Prime in 60 Seconds

	Brute-force	Smart Brute-force	Fermat	Miller-Rabin
Test 1	146539	10008881	14577019	8920403
Test 2	145283	9601829	14899669	8952347
Test 3	144563	9943949	14923903	8892509

4.2 Largest Mersenne Prime Verified in 120 Seconds

p	Brute-force	Smart Brute-force	Lucas-Lehmer	Fermat	Miller-Rabin
19	$< 1\ s$	$< 1\ s$	$< 1\ s$	$< 1\ s$	$< 1\ s$
31	$276.11\ s$	$< 1\ s$	$< 1\ s$	$< 1\ s$	$< 1\ s$
61	-	$100.26\ s$	$< 1\ s$	$< 1\ s$	$< 1\ s$
107	-	$> 300\ s$	$< 1\ s$	$< 1\ s$	$< 1\ s$
607	-	-	$< 1\ s$	$< 1\ s$	$< 1\ s$
1279	-	-	$< 1\ s$	$< 1\ s$	$< 1\ s$
4423	-	-	$< 1\ s$	$< 1\ s$	$2.72\ s$
9689	-	-	$2.04\ s$	$2.49\ s$	$24.80\ s$
11213	-	-	$3.12\ s$	$3.85\ s$	$37.71\ s$
19937	-	-	$16.42\ s$	$20.35\ s$	$198.63\ s$
23209	-	-	$25.65\ s$	$31.37\ s$	-
44497	-	-	$173.61\ s$	$208.85\ s$	-
86243	-	-	-	-	-

5 Discussion

As showed in both tests, the Brute-force was the slowest of all algorithms tested, and could only handle numbers up to the magnitude of 10^9. This makes sense considering the time complexity, $\mathcal{O}(N)$. Since a modern computer can do a little more than 10^8 operations per second, a result of 280 seconds is expected. For the algorithm to verify $2^{61}-1$, it would take around 10^{10} seconds, which is more than 300 years. This demonstrates the fact that efficient algorithms must be found. As $\mathcal{O}(N)$ grows linearly, it does not take much for the runtime to exceed our lifetimes.

The Smart Brute-force, with its time complexity of $\mathcal{O}(\sqrt{N})$, was considerably faster and completed $p = 61$ in 100 seconds, rather than 300 years as in the case of the normal brute-force. However, that still doesn't increase the algorithm's performance by much, since it exceeds the time-limit on the next test, $p = 107$.

On the other hand, Lucas-Lehmer could handle $2^{44,497}$, which is approximately equal to $10^{13,394}$. Compared to the Brute-force algorithms, this is a substantial step for the deterministic algorithms. Lucas-Lehmer demonstrates how efficient an algorithm can become if it exploits properties of Mersenne primes.

WRITE ABOUT FERMAT AND MILLER-RABIN HERE

Overall, the algorithms tested correlate well to their respective Big O notation. However, there is still a non-negligible variance. This is especially shown in test (4.1) where all of the different sub-tests for each algorithm yielded different numbers. The explanation for this phenomena is that the performance of a computer is not constant. The change of CPU temperature, clock speed and prioritisation of processes within the operating system, all lead to change in performance. These factors are all beyond the control of the user. In hindsight, it would be wise to have several sub-tests in (4.2)

as-well, since it was clearly shown that performance fluctuates over time, and that the tests done in (4.2) therefore are not reliable. This is something that should be done if this study were to ever be re-done.

Even though the deterministic algorithms managed to verify numbers up to $10^{13,394}$, it's nothing compared to the largest prime known to man, $2^{82,589,933} - 1$ [11]. There are many factors as to why the results don't come near the largest prime. One of the factors being that the algorithms chosen for this study are both simple to understand and implement. This is done to satisfy the purpose earlier introduced. The algorithms used in finding the largest primes are *extremely* complicated to understand and especially implement. Other factors can be time and computers, which are huge limits for this study.

The most efficient algorithm which was tested by this study was the Lucas-Lehmer algorithm, as shown in (4.2). It could handle $2^{44,497} - 1$. This is however nowhere near the current largest prime, $2^{82,589,933} - 1$. There are several different reasons for this. The current largest prime was not even found using the Fermat primality test. The algorithm used to find this large number is a more optimised version of the Lucas-Lehmer. This version is extremely complicated and is far beyond the scope of this study, since it requires a ph.D. to even understand. Even if the Fermat primality test were used to find this large prime, we would not be able to recreate it. The laptop used in this study is nothing compared to the GIMPS coalition of many computers, each one of them more powerful than this laptop.

A future test regarding this subject could be improved by testing different values for the constant k and exploring how these different values would affect the runtimes of the probabilistic algorithms. Another area of improvement would be to increase the number of test cases in both (4.1) and (4.2) in order to 1. draw conclusions regarding the variance and 2. draw conclusions regarding the accuracy of the runtimes in test (4.2). Only having one test case per algorithm and p is like conducting a survey with one test person, it is not accurate at all.

The constant k was chosen to be 10, which is a relatively arbitrary number. Research on the internet did not yield a definite answer as to what value k should have. Some algorithms had $k = 3$ while others had $k = 10$. The run times of the probabilistic algorithms are proportional to this k constant. Therefore, testing different values of k for each algorithm would be essential to determine the actual efficiency of the probabilistic algorithms. Another aspect of having an arbitrary k is that the accuracy of the tests might be impeded. No control was made to make sure that the algorithms actually returned a true boolean value. This would also be a great area of improvement, as this fact basically deems these tests as useless.

6 Conclusion

To conclude, the fastest algorithm was Lucas-Lehmer, closely followed by the Fermat primality test. Lucas-Lehmer averaged better since it specialises in Mersenne primes, and therefore larger numbers. It is hard to determine which algorithm was most efficient because of this reason. The algorithms specialise in something different, i.e. the Fermat primality test was fast, with the drawback of being probabilistic. Fermat and Miller-Rabin also depend on the different values of k, giving an error in the methods; *should one sacrifice accuracy for speed*?

The study could not get near the largest prime number. The three factors for this being:

1. The **algorithms** chosen were not efficient enough, but rather to be simple
2. **Computational** power was lacking
3. There wasn't enough **time**

The study accomplished its goal, to give an introduction to applied programming and number theory. It is not intended for scientists, mathematicians or experts in this field, instead it it should be used to introduce students or interested individuals to the aforementioned subjects.

7 Acknowledgements

We are sincerely thankful to these dear mentors of ours. These individuals have taught us the required mathematics to perform a study of this grade, while at the same time inspiring us to seek our potential.

A thank you to:

Björn Norén for teaching us advanced mathematics and to always look after our health. The removal of the chocolate from our coffees has made us considerably more manly. We inform you that the chest hair has grown considerably during the span of this course.

Jonas Ingesson for making the subject entertaining in a comedic way, an excellent way to engrave hard-to-memorise topics. We appreciate that you take care of your magnificent sheep instead of correcting our exams that our lives depend on.

Belal Tulimat for always pushing us to our highest level, giving us challenge after challenge to ascend our brains to the heavens. By taking our pens and showing insane solutions our brains have been infused with the ultimate calculus of doom.

Last but not least, **Bertil Nilsson**, the great lecturer at Halmstad University that has taught us that computers have the potential to solve every mathematical problem in blink of an eye. We thank you for many Mathematica-induced headaches.

8 References

[1] Jim Anderson. *Python vs C++: Selecting the Right Tool for the Job.* https://realpython.com/python-vs-cpp/. Information taken 26 April, 2020.

[2] Python Software Foundation. *Python.* https://www.python.org/. Information taken 3 January, 2020.

[4] Inc. Mersenne Research. *Great Internet Mersenne Prime Search.* https://www.mersenne.org/. Information taken 2 April, 2020.

[5] Inc. Wikimedia Foundation. *Big O notation.* https://en.wikipedia.org/wiki/Big_O_notation. Information taken 3 January, 2020.

[6] Inc. Wikimedia Foundation. *C++.* https://en.wikipedia.org/wiki/C%2B%2B. Information taken 26 April, 2020.

[7] Inc. Wikimedia Foundation. *Extended Riemann hypothesis.* https://en.wikipedia.org/wiki/Generalized_Riemann_hypothesis#Extended_Riemann_hypothesis_(ERH). Information taken 9 March, 2020.

[8] Inc. Wikimedia Foundation. *Fermat primality test.* https://en.wikipedia.org/wiki/Fermat_primality_test. Information taken 24 February, 2020.

[9] Inc. Wikimedia Foundation. *Fermat's little theorem.* https://en.wikipedia.org/wiki/Fermat%27s_little_theorem. Information taken 9 March, 2020.

[10] Inc. Wikimedia Foundation. *Fundamental theorem of arithmetic.* https://en.wikipedia.org/wiki/Fundamental_theorem_of_arithmetic. Information taken 28 December, 2019.

[11] Inc. Wikimedia Foundation. *Largest known prime number.* https://en.wikipedia.org/wiki/Largest_known_prime_number. Information taken 27 December, 2019.

[12] Inc. Wikimedia Foundation. *Lucas–Lehmer primality test.* https://en.wikipedia.org/wiki/Lucas%E2%80%93Lehmer_primality_test. Information taken 26 February, 2020.

[13] Inc. Wikimedia Foundation. *Miller–Rabin primality test*. https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test. Information taken 24 February, 2020.

[14] Inc. Wikimedia Foundation. *Prime number theorem*. https://en.wikipedia.org/wiki/Prime_number_theorem. Information taken 3 January, 2020.

[15] Inc. Wikimedia Foundation. *RSA (cryptosystem)*. https://en.wikipedia.org/wiki/RSA_(cryptosystem). Information taken 25 April, 2020.

[16] Inc. Wikimedia Foundation. *Time complexity*. https://en.wikipedia.org/wiki/Time_complexity. Information taken 3 January, 2020.

[17] Inc. Wikimedia Foundation. *Wikipedia*. https://en.wikipedia.org/wiki/Main_Page.

9 Appendix

[3] GitHub. *Primes GA*. https://github.com/FlySlime/Primes-GA.