Halmstad

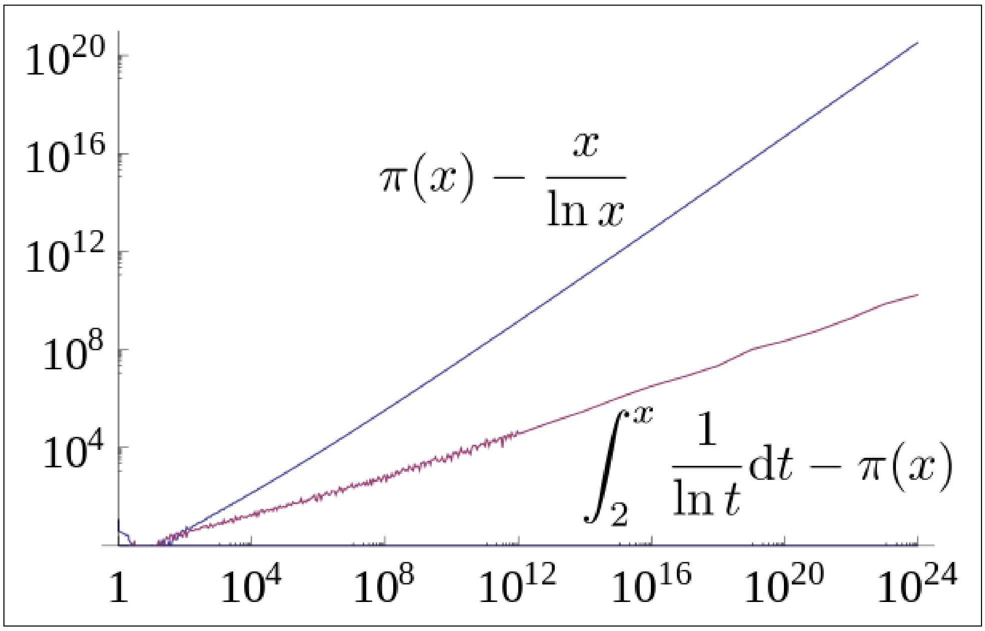
KATTEGATTGYMNASIET

Prime Numbers

Understanding How the Largest Primes Came to Be

Ibrahim Abdulhussein,
Robin Brusbo

April 20, 2020



[1]

[1] The Prime Number Theorem [13]

Abstract

hej ibra du är finnnn

Contents

1 Introduction

1.1 History

Prime numbers have been studied since Ancient Greece, and have, perhaps, been known to us for even longer. [12]

1.2 Aim

The purpose of this essay is to study how large prime number are found. Investigating all the aspects of prime numbers would be a near impossible task; therefore, this essay will focus more on learning about the methods and algorithms of finding new primes. The study will also present ways to prove that a given number, n, is indeed prime.

The relevant questions for this study are the following:

1. What methods can be used to find new primes?
2. What are some algorithms for primality testing?
3. Which algorithm is the most efficient and why?

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. An exception to this definition is 1, since it's the first natural number.

If a number, n, is not prime, it is referred to as a *composite 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 127 which is a prime. Apart from having a connection to perfect numbers, mersenne primes are useful when it comes to finding larger and larger primes, because of their special properties which the computer programs exploit. This is undoubtedly the reason the eight largest primes are all mersenne primes.

2.3 The Fundamental Theory of Arithmetic

The Fundamental Theory of Arithmetic [7] 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 factorization containing only primes, unless it is a prime itself.

Important to know is that there is an infinite amount of primes. The proof is a quite easy by contradiction, but nonetheless beautiful:

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

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

Let the constant Q be the product of all the primes in the list and add 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 [13] 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.

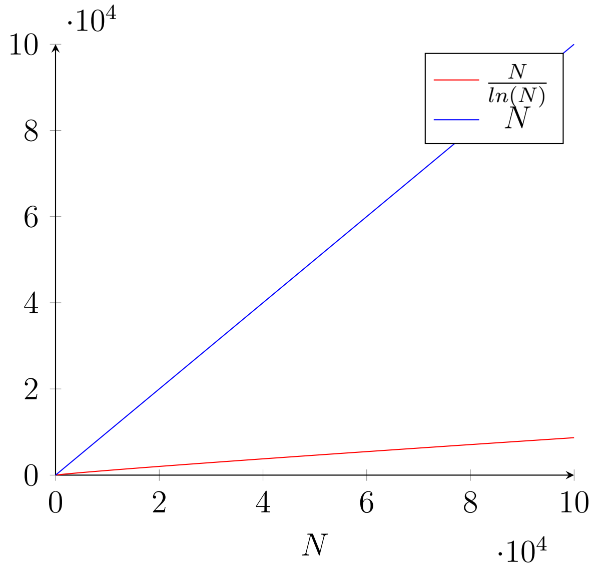


Figure 1: The graph of $\pi(N)$ and N from 0 to 10^5 letting us compare the relationship between the number N and the approximate amount of primes below it.

This proves that primes do not show up linearly, meaning a computer that is twice as powerful will *not* produce twice as many primes. Instead, the most important and crucial part of generating and verifying primes is the optimization of the *algorithms*.

2.5 Time Complexity

Time complexity [15] is a concept within computer science, which describes the approximate time for a program to complete. The study will make heavy use of the Big O Notation [3], which notates how the run time increases as the input size increases. For example, $O(N)$ 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 $O(10N)$. On the other hand, $O(log(n))$ grows logarithmically, which is far more efficient for bigger input sizes, as $O(log(N))$ 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 by this study.
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 $O(N)$ and another with $O(N^2)$, will have an overall time complexity of $O(N^2)$ since N^2 dominates for large values of N

The Big O Notation will be used to determine whether an algorithm with a large number, n, will succeed or run for a *very long time*[2]. Considering that the largest known prime is $24,862,048$ digits [8], algorithms have to be efficient to perform a primality test.

2.5.1 Examples of Big O Notation

A program that runs in $O(N)$ would be a function that inputs an integer N and outputs every number up to N. This runs in $O(N)$ time because it does

[2] Some programs will not finish until the sun explodes, which is quite impractical.

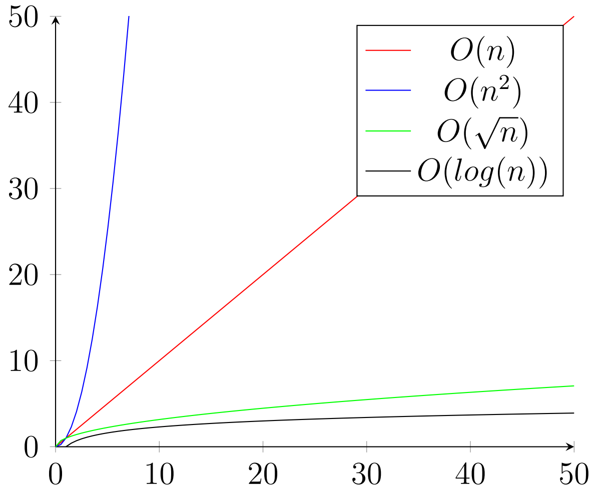


Figure 2: The graph does a lot of stuff. The graph does a lot of stuff. The graph does a lot of stuff. The graph does a lot of stuff. The graph does a lot of stuff. The graph does a lot of stuff. The graph does a lot of stuff.

$1 + N$ operations which results in $O(n)$:

```
def linearTime(N):
    for number in range(N):
        print(number)
```

Since the program runs in $O(N)$ time, increasing the input by a factor of 10 would also increase the number of operations done by a factor of 10.

The second example is a program that runs in quadratic time, i.e. $O(N^2)$. Compared to the first example, this program will print every number up to N, N times.

```
    def quadraticTime(N):
        for number in range(N):
            for number in range(N):
                print(number)
```

The third program describes $O(\sqrt{n})$. It is a simple program that sums all numbers up to $\sqrt{N}$.

```
from math

def sqrtTime(N):
    sum = 0
    for number in range(math.isqrt(N)):
        sum += number
    print(sum)
```

The fourth and last example will be about $O(log(n))$. The algorithm that is used in this example is called *Binary Search* [4]. It's an algorithm used for quickly guessing a number between, for example, 1 and 100. Usually two players are required to play this guessing game, but with a computer the user will give an input that the computer will try to guess. The output will be the amount of "guesses" the algorithm performed.

```
def binarySearch():
    begin = int(input("The number will be between _ and _
                                   \n"))
    end = int(input())
    value = int(input("What value will you input?\n"))
    guesses = 0

    while True:
        guesses += 1
        mid = int((begin + end) / 2)
        if mid > value:
            end = mid - 1
        elif mid < value:
            begin = mid + 1
        else:
            break

print(guesses)
```

2.6 Deterministic vs. Probabilistic

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

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 or 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 is composite, not all numbers below it will divide it. The accuracy of the test, i.e. the probability that it returns a correct answer, could be increased by increasing the amount of unique factors tested. This will be an important fact in the Fermat and Miller-Rabin primality tests. There, the variable k will be used to specify the number of rounds the test will run.

2.7 Algorithms

2.7.1 Brute-force

The brute force method is arguably the simplest of all algorithms. This algorithm tests all the numbers, n, between 2 and $p - 1$ and checks if n satisfies $n \equiv 0 \pmod{p}$, which makes n a divisor of p and therefore p is definitely not prime. If the loop, however, completes without finding any divisors of p, then p is definitely prime. The pros of this method are that it is simple to understand, and that it is a definite test; p is prime if and only if there are no divisors except for 1 and p itself. The big con is that this algorithm is slow. It runs in $O(N)$ time which is the slowest of all algorithms which will be tested in this essay.

```
    Insert code here
```

2.7.2 Smart Brute-force

The smart brute force is also a brute force algorithm, but it utilizes some of the properties of primes. Since a prime can never have the factor 2 in them, it is sufficient to only test 2 and the odd numbers. On top of this, the smart brute force only tests numbers up to and including $\sqrt{p}$. This is because, assuming that p is composite, it must have at least 2 factors. If there exists a factor larger than $\sqrt{p}$, it must have already been tested. This algorithm will run in $O(\sqrt(n))$ time, which is much faster than the original brute force.

```
    Insert code here
```

2.7.3 Lucas-Lehmer

The Lucas-Lehmer algorithm is a deterministic primality test that only works for Mersenne numbers. It takes advantage of a special property of Mersenne numbers:

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

There exists a proof for this, which will not be covered by this essay.

The Lucas-Lehmer test has helped the GIMPS (Great Internet Mersenne Prime Search) to find many of the largest primes known to man. This because the time complexity of this test is much faster compared to the other tests. Another advantage it has over some of the other tests, is that it is deterministic.

2.7.4 Fermat

2.7.5 Miller–Rabin

3 Materials and Methods

3.1 Resources

The most important resource to be used in this study will be Wikipedia and the countless articles written regarding this subject. The programming language used to implement and time the algorithms will be Python[3] [1].

As of today, the largest primes are found using supercomputers running these algorithms. Using a supercomputer is not plausible for this study, therefore the maximum limit for primes will be 10^{14}.

3.2 Methods

During the primality testing [11] there will be two test cases for the algorithms. The first one will determine if n is a prime or composite, and the second one will generate the largest prime within a time limit. At the start of the code a driver function will be defined to serve as the ”starter” code for the program - deciding what test to perform. The driver function will call a primality testing function called **primeTest** which will return a Boolean[4] value.

Two groups of algorithms will be tested: the deterministic and the probabilistic tests. The driver function will be called to start a timer and execute one of the following algorithms:

1. Deterministic tests [5]
   (a) Brute-force
   (b) Smart Brute-force
   (c) Lucas-Lehmer [9]
2. Probabilistic tests [14]
   (a) Fermat Primality [6]

[3] Although Python is one of the slowest languages, the simplicity of it makes it an invaluable tool for this study.

[4] A function that returns *True* or *False*

(b) Miller–Rabin [10]

The function will be called upon five times to average an ”accurate” runtime. The same integers will be used across all tests. These runtimes are noted and later compared in the discussion. The following Python code describes the setup:

```
import time
# from FILE import ALGORITHM

def driver():
    numbers = [...] # Values to be appended
    times = []
    for n in numbers:
        start = time.time()
        primeTest(n)
        runTime = time.time() - start
        times.append(runTime)
    print(sum(times) / len(times))

def primeTest(n):
    if ALGORITHM(n): # Algorithm function
        return True
    return False

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

The algorithms will be stored in a separate file, hence the *from FILE import ALGORITHM* at the top of the code. To see further details and the ∗*.py* files, refer to the essays GitHub page [2].

To generate the largest prime within 30 seconds, the **primeTest** function will remain the same, however the driver function will receive some changes:

```
def driver():
    n = 2
```

```
primes = []
timelimit = time.time() + 30  # Thirty seconds limit
while (time.time() < timelimit):
    if primeTest(n):
        primes.append(n)
    n += 1
print(max(primes))
```

4 Results

4.1 Finding largest prime in 60 seconds

TO BE REMOVED: Three tests are performed to try to eliminate factors. The median is taken from the results.

	Brute-force	Smart Brute-force	Fermat	Miller-Rabin
Test 1	146539	6431833	14577019	8920403
Test 2	145283	6323773	14899669	8952347
Test 3	144563	6421421	14923903	8892509

4.2 Biggest Mersenne prime tested in 120 seconds

TO BE REMOVED: At least two decimal numbers O if success X if failure - if 5 minute over, manually cancel the program.

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	-	🟩 $101.54\ s$	🟩 $< 1\ s$	🟩 $< 1\ s$	🟩 $< 1\ s$
107	-	-	🟩 $< 1\ s$	🟩 $< 1\ s$	🟩 $< 1\ s$
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	-	-	🟥 $172.51\ s$	🟥 $208.85\ s$	-
86243	-	-	-	-	-

5 Discussion

6 Conclusion

7 Acknowledgments

Belal Tulimat The man, the myth, the legend. Our very own private teacher. With your special ability to steal pens from our hands, you have taught us everything we need and don't need to know about hyperbolas, integrals and the Corona virus.

Loke Gustafsson The one and only Loke. There is no word in the English dictionary that can describe you, so the only correct word would be "Lokeism". The fact that the you thought our work was good, really gave us the motivation to continue.

Björn Norén Our favourite bald-headed teacher that gave us way too high grades in physical education. This was the only reason we chose this class. If we were able to get too high grades in PE, why wouldn't we be able to succeed in math?

Jonas Ingesson

Fredrik Larvall

Christiane Miller Arguably the most warm-hearted teacher on earth. You always do your best for your students.

8 References

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

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

[4] Wikipedia. *Binary Search*. https://en.wikipedia.org/wiki/Binary_search_algorithm. Information taken January 13, 2020.

[5] Wikipedia. *Deterministic algorithm*. https://en.wikipedia.org/wiki/Deterministic_algorithm. Information taken January 17, 2020.

[6] Wikipedia. *Fermat primality test*. https://en.wikipedia.org/wiki/Fermat_primality_test. Information taken February 24, 2020.

[7] Wikipedia. *Fundamental theorem of arithmetic*. https://en.wikipedia.org/wiki/Fundamental_theorem_of_arithmetic. Information taken December 28, 2019.

[8] Wikipedia. *Largest known prime number*. https://en.wikipedia.org/wiki/Largest_known_prime_number. Information taken December 27, 2019.

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

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

[11] Wikipedia. *Primality test*. https://en.wikipedia.org/wiki/Primality_test. Information taken January 7, 2020.

[12] Wikipedia. *Prime Number*. https://en.wikipedia.org/wiki/Prime_number. Information taken December 5, 2019.

[13] Wikipedia. *Prime number theorem*. https://en.wikipedia.org/wiki/Prime_number_theorem. Information taken January 3, 2020.

[14] Wikipedia. *Randomized algorithm*. https://en.wikipedia.org/wiki/Randomized_algorithm. Information taken January 17, 2020.

[15] Wikipedia. *Time complexity*. https://en.wikipedia.org/wiki/Time_complexity. Information taken January 3, 2020.

9 Appendix

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