

Prime Numbers

Understanding How the Largest Primes Came to Be

Ibrahim Abdulhussein,
Robin Brusbo

April 20, 2020



¹The Prime Number Theorem [\[13\]](#)

Abstract

Contents

1	Introduction	3
1.1	History	3
1.2	Aim	3
2	Theory	4
2.1	What Is a Prime?	4
2.2	The Fundamental Theory of Arithmetic	4
2.3	The Prime Number Theorem	5
2.4	Time Complexity	5
2.4.1	Examples of Big O Notation	6
3	Materials and Methods	9
3.1	Resources	9
3.2	Methods	9
3.3	Algorithms	11
3.3.1	Brute-force	11
3.3.2	AKS Primality Test	12
3.3.3	Fermat Primality	12
3.3.4	Miller–Rabin	12
4	Results	14
4.1	Primality Testing	14
4.2	Generating Largest Prime	14
5	Discussion	15
6	Conclusion	16
7	Acknowledgments	17
8	References	18
9	Appendix	20

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. Primes are without a doubt beautiful mathematically, but what real life application do primes have?

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 The Fundamental Theory of Arithmetic

The Fundamental Theory of Arithmetic [8] 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:

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

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

$$Q = p_1 * p_2 * p_3 * \dots + 1$$

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.

2.3 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 .

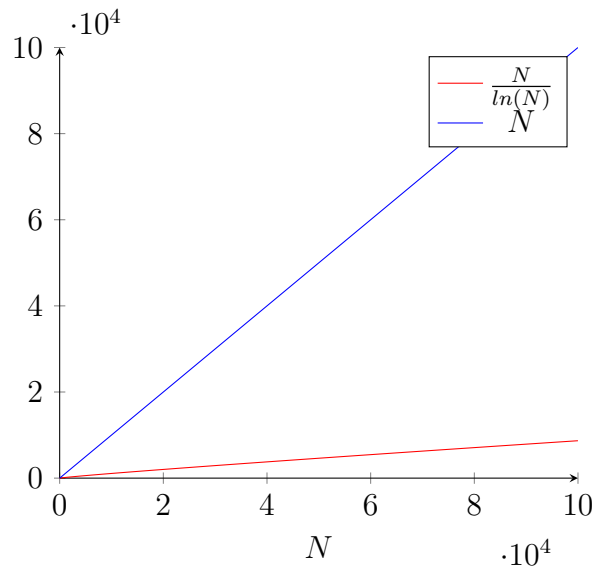


Figure 1: The graph of $\pi(N)$ and N from 0 to 10^5 letting us compare the relationship between the number N and the approximate amount of primes below it.

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*.

2.4 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 [4], 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 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 [9], algorithms have to be efficient to perform a primality test.

2.4.1 Examples of Big O Notation

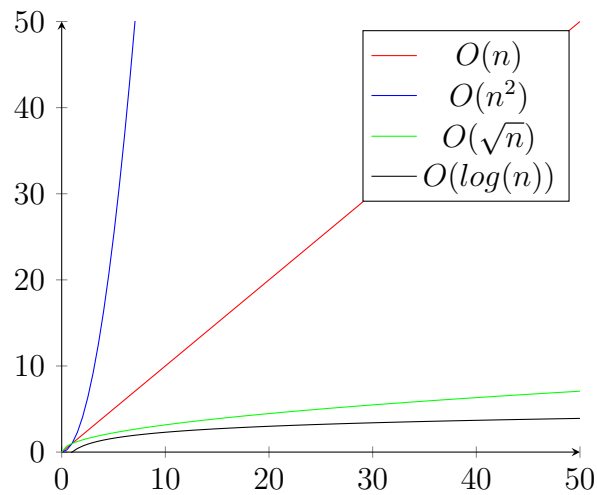


Figure 2: The graph does a lot of stuff. The graph does a lot of stuff. The graph does a lot of stuff. The graph does a lot of stuff. The graph does a lot of stuff. The graph does a lot of stuff. The graph does a lot of stuff.

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

A program that runs in $O(N)$ would be a function that inputs an integer N and outputs every number up to 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* [5]. 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)
```

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 Python³ [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 boolean⁴ 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 [6]
 - (a) Brute-force
 - (b) Smart Brute-force
 - (c) AKS Primality Test [3]
2. Probabilistic tests [14]
 - (a) Fermat Primality [7]

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

⁴A 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))

```

3.3 Algorithms

Vi skriver lite trivia och importance om algoritmer

3.3.1 Brute-force

This is the Brute-force algorithm.

```

def brute(n):
    for x in range(2, n):
        if n % x == 0:
            return False
    return True

```

This is the Smart Brute-force algorithm.

```

import math

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

```

3.3.2 AKS Primality Test

3.3.3 Fermat Primality

```
import random

def fermat(n, k):
    # Use _ as "index" variable - means no declared
    # variable
    for _ in range(k):
        a = random.randint(2, n - 2)
        if pow(a, n - 1, n) != 1:
            return("composite")
        return("probably prime")
```

3.3.4 Miller–Rabin

```
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):
    r, d = findIntegers(n)
    # Use _ as "index" variable - means no declared
    # variable
    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 "composite"
    return "composite"
return "probably prime"
```

4 Results

4.1 Primality Testing

Brute-force	Smart brute-force	Sieve of Eratosthenes
0. <i>x</i> seconds	0. <i>x</i> seconds	0. <i>x</i> seconds
0. <i>x</i> seconds	0. <i>x</i> seconds	0. <i>x</i> seconds
0. <i>x</i> seconds	0. <i>x</i> seconds	0. <i>x</i> seconds
0. <i>x</i> seconds	0. <i>x</i> seconds	0. <i>x</i> seconds
0. <i>x</i> seconds	0. <i>x</i> seconds	0. <i>x</i> seconds

Table 1: The runtime for each algorithm three times

AVERAGE 1	AVERAGE 2	AVERAGE 3
-----------	-----------	-----------

Table 2: The summed averages of the above results

4.2 Generating Largest Prime

5 Discussion

6 Conclusion

7 Acknowledgments

Loke Gustafsson

The one and only Loke. Finding a word to describe you is extremely hard since there really are no fitting words in the English dictionary. Therefore, the most accurate word would be "Lokeism".

Björn Noren

Jonas Ingensson

Christiane Miller

8 References

- [1] Python Software Foundation. *Python*. <https://www.python.org/>. Information taken January 3, 2020.
- [3] Wikipedia. *AKS primality test*. https://en.wikipedia.org/wiki/AKS_primality_test. Information taken January 17, 2020.
- [4] Wikipedia. *Big O notation*. https://en.wikipedia.org/wiki/Big_O_notation. Information taken January 3, 2020.
- [5] Wikipedia. *Binary Search*. https://en.wikipedia.org/wiki/Binary_search_algorithm. Information taken January 13, 2020.
- [6] Wikipedia. *Deterministic algorithm*. https://en.wikipedia.org/wiki/Deterministic_algorithm. Information taken January 17, 2020.
- [7] Wikipedia. *Fermat primality test*. https://en.wikipedia.org/wiki/Fermat_primality_test. Information taken February 24, 2020.
- [8] Wikipedia. *Fundamental theorem of arithmetic*. https://en.wikipedia.org/wiki/Fundamental_theorem_of_arithmetic. Information taken December 28, 2019.
- [9] Wikipedia. *Largest known prime number*. https://en.wikipedia.org/wiki/Largest_known_prime_number. Information taken December 27, 2019.
- [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>.