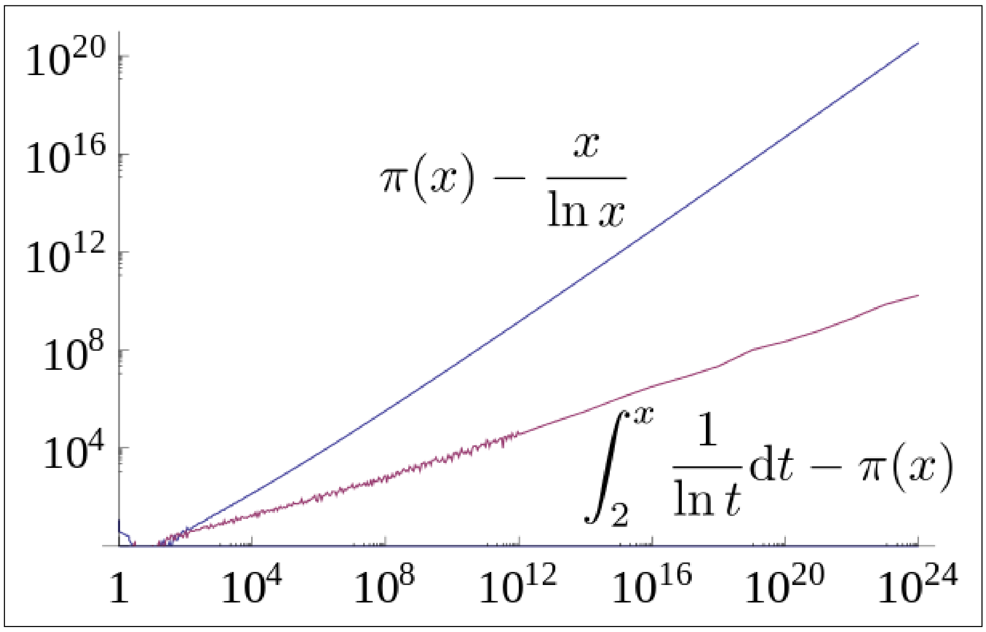
KATTEGATTGYMNASIET

Prime Numbers

Understanding How the Largest Primes Came to Be

Ibrahim Abdulhussein,
Robin Brusbo

April 20, 2020



1

[1]The Prime Number Theorem [9]

Abstract

Contents

1 Introduction

1.1 History

Prime numbers have been studied since Ancient Greece, and have, perhaps, been known to us for even longer. [8] Add more information

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. If the number, n, is not prime, it is referred to as a *composite number.*

An exception to this definition is 1, since it's 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, as of writing this, are mersenne primes [6]. The reason being because of their easily exploitable properties by computers, making them faster to primality test than normal integers.

2.3 The Fundamental Theory of Arithmetic

The Fundamental Theory of Arithmetic [5] 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.

For this study to be relevant, there has to be an infinite amount of primes. There is an easy proof by contradiction for infinite primes:

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:

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

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 [9] 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.

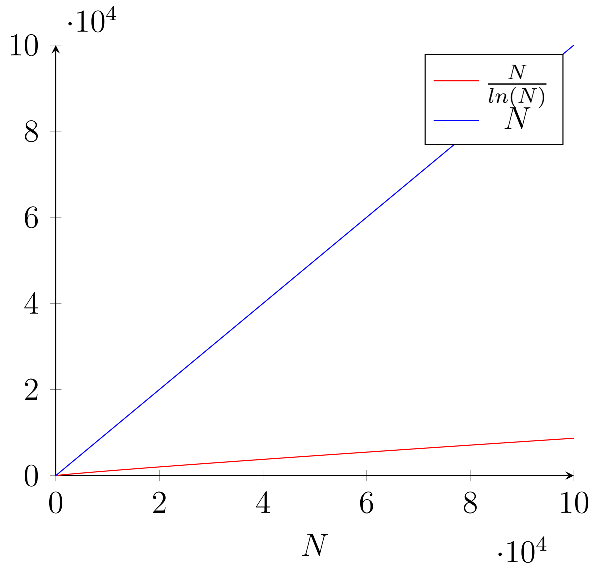


Figure 1: The graph of $\pi(N)$ and N from 0 to 10^5 can be used to compare the relationship between the number N and the approximate amount of primes below it.

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 largest factor for verifying primes faster are the *algorithms*.

2.5 Time Complexity

Time complexity [10] 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 [1], 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.

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 [6], algorithms have to be efficient to perform a primality test.

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

2.5.1 Examples of Big O Notation

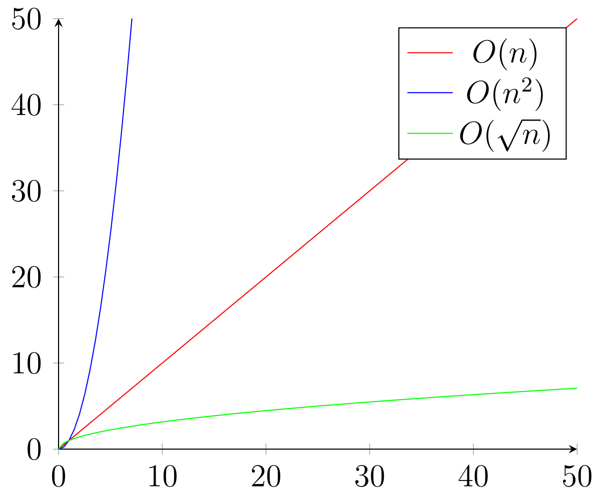


Figure 2: The graph does a lot of stuff. The graph does a lot of stuff. The graph does a lot of stuff. The graph does a lot of stuff. The graph does a lot of stuff. The graph does a lot of stuff. The graph does a lot of stuff.

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 $1 + N$ 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, $O(N^2)$. 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 $\sqrt{N}$.

```
import math

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

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 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, 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 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 $O(N)$, meaning it increases linearly.

```
Insert code here
```

2.7.2 Smart Brute-force

The smart brute force is a variant of the brute force algorithm (2.7.1). By utilizing some properties of primes, the smart brute force has a time complexity of $O(\sqrt{n})$ compared to the $O(N)$ complexity from the brute force.

Since a prime can never have the factor 2 in them, it is sufficient to only test odd numbers (an if statement is written before to loop to test if it has the factor 2). 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

The Fermat primality test [3] is a probabilistic algorithm that resembles the Miller-Rabin test. The time complexity for the algorithm is $O(klog^2nloglognlogloglogn)$. Although the long notation for the algorithm, the speed of the test can be compared to Lucas-Lehmer.

Fermat's little theory [4] states *"If p is prime and a is not divisible by p then: "*

$$\boxed{a^{p-1} \equiv 1 \pmod{p}}$$

2.7.5 Miller–Rabin

The Miller-Rabin primality test [7], first discovered by Russian mathematician M. M. Artjuhov in 1967, is an algorithm that has two versions. 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 [2]. However, the probabilistic version is unconditional, meaning that it does not depend on an unproven theorem. This makes the probabilistic version reliable. If the extended Riemann hypothesis were to be disproved, the deterministic version would fall apart. The probabilistic version is also faster for bigger n compared to the deterministic version. The probabilistic is therefore chosen for this study.

The Miller-Rabin primality test is considered to be one of the fastest algorithms to verify if an n is prime or not. The time complexity for the probabilistic version is $O(klog^3(n))$, where k is the amount of rounds the algorithm will perform.

insert how it works here

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 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. 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 one of the slower programming languages, around 10 times slower than $C{+}{+}$, the simplicity and ease of understanding outweighs the slowness.

The tests will be performed on an 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 it is important that all tests are 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 programs will be given 60 seconds to loop through all 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 item 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 algorithms that will be tested here are the following: Brute-Force, Smart Brute-Force, Fermat and Miller-Rabin. The reason Lucas-Lehmer will not be tested here is that it only works on Mersenne numbers, and not all natural numbers are Mersenne numbers.

The second test, called "Biggest Mersenne prime tested in 120 seconds", will test Mersenne primes for primality. Of course, all algorithms should return true, but that is not what is interesting. What will be looked at are the run-times, 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 finishes past 120 seconds, or if it does not finish at all withing a reasonable amount of time, i.e. around 5 minutes, it is deemed a "failure". The numbers being tested are the following: 19, 31, 61, 107, 607, 1279, 4423, 9689, 11213, 19937, 23209, 44497, 86243. Note that these numbers are the exponents of the Mersenne prime being tested, and not actually the numbers to test. In this test, all the algorithms will be compared, since all algorithms are able to test mersenne primes. For the probabilistic tests, the constant k will be equal to 10, an arbitrary number to decide the number of rounds. Why 10 was chosen, is discussed in the discussion.

In order to conduct these tests, a "driver" function will be used to call the functions and take the time. The code for the different algorithms will be translated from pseudocode into Python by us. The code of the algorithms and driver function will not be written in this document, however they can be found in this essays GitHub repository. The link is in the appendix.

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	10008881	14577019	8920403
Test 2	145283	9601829	14899669	8952347
Test 3	144563	9943949	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	-	$100.26\ s$	$< 1\ s$	$< 1\ s$	$< 1\ s$
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	-	-	-	$208.85\ s$	-
86243	-	-	-	-	-

5 Discussion

Answeer questions Relate to time complexities Why k = 10? Were the answers expected? How could a future test be improved? Why could we not find the biggest mersenne prime? What order of magnitude are the numbers on? Why did the results in test 1 differ so much? What could our tests imply? Infuse results with meaning. Brute force will not finish until sun explodes

6 Conclusion

7 Acknowledgments

8 References

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

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

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

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

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

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

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

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

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

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

9 Appendix