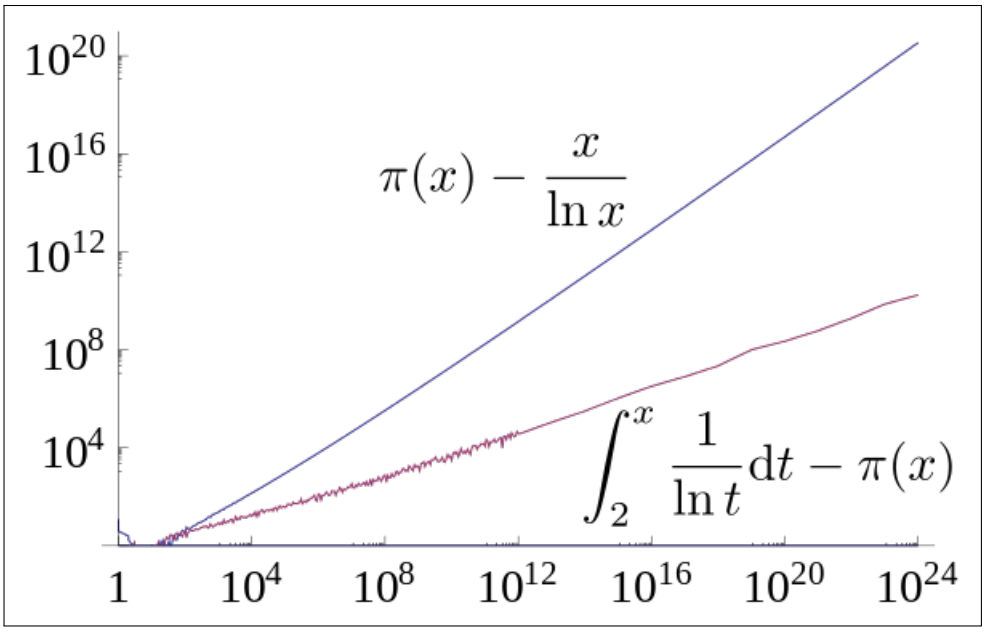
Halmstad
KATTEGATTGYMNASIET

Prime Numbers

Understanding How the Largest Primes Came to Be

Ibrahim Abdulhussein,
Robin Brusbo

April 20, 2020



1

[1]The Prime Number Theorem [11]

Abstract

Prime numbers are a fundamental part of number theory. They have many important properties and it is therefore of some number theorists interests to find larger and larger primes. In order to accomplish this, efficient algorithms have to be developed. This study investigates the efficiency of five common primality testing algorithms: Brute-force, Smart Brute-force, Lucas-Lehmer, Fermat and Miller-Rabin. Two different tests were performed to compare the algorithms.

In the first test, the algorithms were given the task of finding the largest prime number within 60 seconds by looping the natural numbers from 2. The first test could not include Lucas-Lehmer, as it only works for mersenne numbers.

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 indeed 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 showed that the Fermat primality test was the fastest algorithm to find primes, albeit probabilistic.

This study has little to no use in actually finding new primes, as much more efficient algorithms are currently used for this purpose. However, this study can be used 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 are a beautiful mathematical concept. Out of all other numbers, primes have had an invaluable importance to our lives. The most common example being RSA-encryption, where a product of two *extremely* large primes are exploited for its toughness to factor. In order to decrypt the message, one would need to have one of the factors.

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 [8]. Using a method that *Checks all numbers below n* is simply not efficient enough for large primes.

The study will focus on efficient 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'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 [8]. 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 [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 factorisation 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: $p_1, p_2, p_3, p_4, p_5, \ldots$

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 [11] 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.

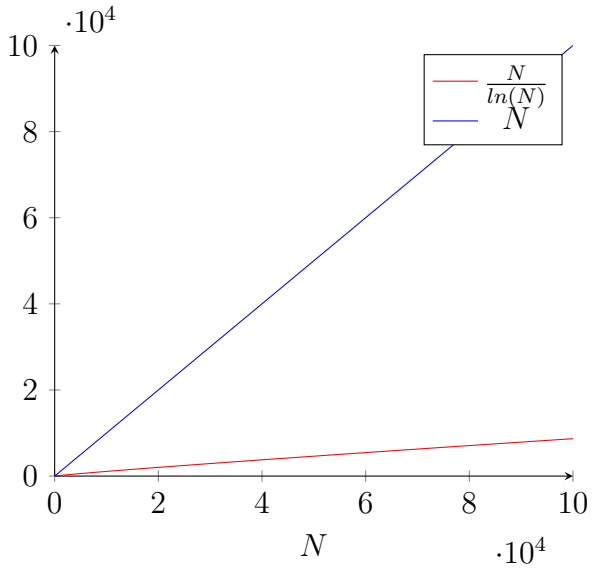


Figure 1: The graph of $\pi(N)$ and N from 0 to 10^5 can be used to compare the relationship between the number N and the approximate amount of primes below it.

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 [12] 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 [3], 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.

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

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 $\mathcal{O}(N)$, 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 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.

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 a factor larger than $\sqrt{p}$ must have already been tested.

```
Insert code here
```

2.7.3 Lucas-Lehmer

The Lucas-Lehmer algorithm [9] 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) [2] to find many of the largest primes known. This because the time complexity of the algorithm is much faster compared to the other algorithms. The time complexity being $\mathcal{O}(\log^2(N) \log\log N)$.

2.7.4 Fermat

The Fermat primality test [5] is a probabilistic algorithm. It's whole concept is based on Fermat's little theory [6] 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.

2.7.5 Miller–Rabin

The Miller-Rabin primality test [10], 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 [4]. 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.

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 slowest programming languages, around 10 times slower than C++, 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 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 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 median will be taken from the results. 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 on Mersenne numbers.

The second test, called "Biggest Mersenne prime tested 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". 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 at the study's GitHub page [1].

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 Biggest Mersenne prime tested 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	-	-	$< 1\ s$	$< 1\ s$	$< 1\ s$
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

As showed in 4.2, 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, $O(n)$. Since a normal computer can do a little more than 10^8 operations per second, a result of 280 seconds is expected. For the algorithm to finish $2^{61} - 1$, it would take around 10^{10} seconds, which is more than 300 years. This demonstrates the fact that efficient algorithms must be found, as $O(n)$ grows linearly, it does not take much for the run time to exceed our lifetimes.

The smart brute-force, with its time complexity of $O(sqrt(n))$, was considerably faster and completed $p = 61$ in 100 seconds, rather than 300 years as in the case of the normal brute-force. However, this does not give the algorithm much hope, because $p = 107$ exceeded the time limit of 120 seconds.

On the other hand, Lucas-Lehmer could handle numbers up to 10^{6986}. This is undoubtedly better than the brute-forces, and shows how important it is to take advantage of the properties of Mersenne primes in order to make a much more efficient deterministic algorithm.
Even though our results showed that these relatively simple algorithms could handle numbers on the order of magnitude of $10^{13,394}$, this is far from the largest prime known to man. The reason behind this is the limitations of this essay. In order to find larger prime numbers than those tested in this essay, even more efficient algorithms must be used, as well as more powerful computers. However, these algorithms are incredibly hard to even understand, and are even harder to code.

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.

The most efficient algorithm which was tested by this study was the Fermat primality test, 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 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.

6 Conclusion

7 Acknowledgments

8 References

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

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

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

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

[6] Wikipedia. *Fermat's little theorem*. https://en.wikipedia.org/wiki/Fermat%27s_little_theorem. Information taken March 9, 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. *Prime number theorem*. https://en.wikipedia.org/wiki/Prime_number_theorem. Information taken January 3, 2020.

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

9 Appendix

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