

The Frequency of Mersenne Primes

Notational Legend:

**Definition 1.1:** Mersenne Primes: A prime number, any with no positive divisors other than 1 and itself, of the form The Mersenne prime is where is the nth prime.

General Observations:

**Theorem 1.1:** Let us note that the fact that the definition of a Mersenne prime being one less than a prime power of two is not arbitrary by the following truth:

Proof:

It should be noted that more general number that does not require the power of 2 to be prime is known only as a Mersenne number. The Mersenne number is: However we will remain interested only in the numbers of the form .

. As in, It is still unknown if the set of Mersenne primes is finite or infinite. The Lenstra-Pomerance-Wagstaff conjecture asserts that there are infinitely many of these primes and predicts their order of grown. It is also not known if there are infinitely many Mersenne numbers with prime powers of two that are composites. As in

The first several Mersenne Primes are as follows:

The first counter example, a number of the form is:

It should be noted, that as grows, the frequency of goes down quite rapidly.

Theorems and Properties:

**Definition 1.2**: , the sum of positive devisors of n:

**Definition 1.3:** : The set of perfect numbers. A number is perfect if that number is equal to the sum of its proper positive divisors. Equivalently, These numbers were first referenced in Euclid’s Elements and where called *τέλειος ἀριθμός.*

**Corollary 1.1:**

**Lemma 1.1:** Divisors of Product of Coprime Integers: .

The reader may want to prove that themselves.

**Lemma 1.2:** The Multiplicativity of

Proof:

The divisors of are of the form , are divisors of (L1.2)

**Theorem 1.2:** If one has a Mersenne prime, then is a perfect number, and all perfect numbers are off this form. Therefore:

Proof:

Therefore, the first part of this theorem holds true, and that a Mersenne prime will be able to generate new perfect numbers. Let us not prove that all perfect numbers are of this form.

Thus we can say that , where is some constant that will be shown to be one.

Therefore, all perfect numbers are of this form as well.

**Theorem 1.3:** The Forced Base of Two: Mersenne Primes of the form must have the base be 2. As in, .

Proof:

The Finding of Extraordinarily Large Primes

As of January 2016, the largest known prime number is 274,207,281 − 1, a number with 22,338,618 digits. It was found in January of 2016 by the Great Internet Mersenne Prime Search (GIMPS). GIMPS is a project devoted to the finding of primes, and by the easy nature of computation, Mersenne primes are those that the project focuses on. The project has found a total of fifteen Mersenne primes as of January 2016, thirteen of which were the largest known prime number at their respective times of discovery.

But why? Why are the larges primes we know of the Mersenne form? It comes down to how we check if a number is indeed prime. So, let us visit how prime numbers are checked in the lens of computation. We will be using Java to explore these algorithms. Traditionally, an algorithm like the one below can be used to check if some number, is prime:

**boolean** isPrime(**int** n) {

**if** (n == 2) **return** **true**;

**if** (n%2==0) **return** **false**;

**for**(**int** i=3;i\*i<=n;i+=2) {

**if**(n%i==0)

**return** **false**;

}

**return** **true**;

}

This algorithm runs up through the square root of a number n, counting by 2s as to negate the even numbers. And returning true if there if n is not 0 modulo i in a for loop. There are in fact much more sophisticated ways of finding prime numbers as well. Take this strategy, that uses a regular sieve technique to run through all numbers and eliminate their multiples.

**public** **static** **boolean**[] fillSieveOfPrimes(**int** upThrough) {

**boolean**[] a = **new** **boolean**[upThrough];

Arrays.*fill*(a,**true**);

a[0]=a[1]=**false**;

**for** (**int** i=2;i<a.length;i++)

**if**(a[i])

**for** (**int** j=2;i\*j<a.length;j++)

a[i\*j]=**false**;

**return** a;

}

This algorithm is an ancient sieve strategy that still remains relevant today. A more sophisticated version of this algorithm that utilizes the strength and speed of binary is as follows, yet the mathematics remains the same:

**public** **class** Sieve

{

**private** BitSet sieve;

**private** Sieve() {}

**private** Sieve(**int** size) {

sieve = **new** BitSet((size+1)/2);

}

**private** **boolean** is\_composite(**int** k)

{

**assert** k >= 3 && (k % 2) == 1;

**return** sieve.get((k-3)/2);

}

**private** **void** set\_composite(**int** k)

{

**assert** k >= 3 && (k % 2) == 1;

sieve.set((k-3)/2);

}

**public** **static** List<Integer> sieve\_of\_eratosthenes(**int** max)

{

Sieve sieve = **new** Sieve(max + 1); // +1 to include max itself

**for** (**int** i = 3; i\*i <= max; i += 2) {

**if** (sieve.is\_composite(i))

**continue**;

// We increment by 2\*i to skip even multiples of i

**for** (**int** multiple\_i = i\*i; multiple\_i <= max; multiple\_i += 2\*i)

sieve.set\_composite(multiple\_i);

}

List<Integer> primes = **new** ArrayList<Integer>();

primes.add(2);

**for** (**int** i = 3; i <= max; i += 2)

**if** (!sieve.is\_composite(i))

primes.add(i);

**return** primes;

­­­­­­ }

}

Is this fast? Well, lets experiment using a runner program for the first million primes.

**public** **class** SieveRunner {

**public** **static** **void** main(String[] args) {

Sieve a = **new** Sieve();

System.***out***.println(a.*sieve\_of\_eratosthenes*(1000000));

}

}

Sure, this is a fine way to find prime numbers that still remain relatively small. The primes of the size we are finding now are extraordinary. An algorithm of this design would be nearly indefinitely busy if attempted to check the primality of an extraordinarily large prime. Let us explore how we check these primes.

The Lucas Lehmer Test:

The simple, but rather remarkable test is as follows:

The first few terms in are as follows: . It is sequence in OEIS.

The psuedocode is as follows:

*// Determine if M*p *= 2*p *− 1 is prime*

**Lucas–Lehmer**(p)

**var** s = 4

**var** M = 2*p* − 1

**repeat** p − 2 times:

s = ((s × s) − 2) mod M

**if** s == 0 **return** PRIME **else** **return** COMPOSITE

Implemented in java, would look something like this:

**import** java.math.BigInteger;

**public** **class** Mersenne

{

**public** **static** **boolean** isPrime(**int** p) {

**if** (p == 2)

**return** **true**;

**else** **if** (p <= 1 || p % 2 == 0)

**return** **false**;

**else** {

**int** to = (**int**)Math.*sqrt*(p);

**for** (**int** i = 3; i <= to; i += 2)

**if** (p % i == 0)

**return** **false**;

**return** **true**;

}

}

**public** **static** **boolean** isMersennePrime(**int** p) {

**if** (p == 2)

**return** **true**;

**else** {

BigInteger m\_p = BigInteger.***ONE***.shiftLeft(p).subtract(BigInteger.***ONE***);

BigInteger s = BigInteger.*valueOf*(4);

**for** (**int** i = 3; i <= p; i++)

s = s.multiply(s).subtract(BigInteger.*valueOf*(2)).mod(m\_p);

**return** s.equals(BigInteger.***ZERO***);

}

}

// an arbitrary upper bound can be given as an argument

**public** **static** **void** main(String[] args) {

**int** upb;

**if** (args.length == 0)

upb = 500;

**else**

upb = Integer.*parseInt*(args[0]);

System.***out***.print(" Finding Mersenne primes in M[2.." + upb + "]:\nM2 ");

**for** (**int** p = 3; p <= upb; p += 2)

**if** (*isPrime*(p) && *isMersennePrime*(p))

System.***out***.print(" M" + p);

System.***out***.println();

}

}

This is indeed how GIMPS and other large prime searching programs work. The Lucas-Lehmer test was originally developed by Edourad Lucas in 1846 and improved by Lehmer in the 1930s.