

Optimal Mersenne Primes Search Using Modular Groups

Jabari Zakiya – jzakiya@gmail.com

October 12, 2026

Abstract

Mersenne Primes have the form $M_p = 2^p - 1$. They are very rare. As of October 2024 only 52 have been confirmed, the last being the largest known prime. Traditionally, every sequential p value has been used to compute|test whether $2^p - 1$ is prime, requiring first generating a complete sequential list of candidate primes. Presented here is an optimally efficient and fast process to generate candidate M_p values to test without having to use any primes to directly compute $2^p - 1$, using **Mersenne Prime Generators (MPG)**. With them, we can sequentially generate all possible M_p values starting with a p of some known M_p , without any further input or intervention in the search process.

Introduction

Mersenne Primes are named after Marin Mersenne (1588-1648), a French Catholic priest and polymath who was first to create a list of primes of form $M_p = 2^p - 1$. They are very rare, and currently only 52 values of p have been confirmed to create them. In 1996 the **Great Internet Mersenne Primes Search (GIMPS)** – <https://www.mersenne.org/>) project was created to find them.

GIMPS has found 18 of them, including (as of October 2024) the largest confirmed $M_{136,279,841}$, which is also currently the largest known confirmed prime in general. GIMPS uses a distributed network of volunteers to run its code to test|recheck values generated by different candidate primes. To do this it must first generate a list of sequential known primes to use as candidates to test. This is a daunting task in itself, as a complete sequential list of primes become harder to find|verify as they become larger.

Presented here is a method to find the M_p without having to use any primes to directly compute $2^p - 1$. What I will show is, we can create an optimal **MPG**, of a special form, to find every M_p . They only use the prime p for a known M_p to generate the next larger M_p , which then is used to find the next, etc.

The math is simple and straightforward, and uses the properties and math of **modular groups**, which underlie the simple and powerful framework of **Prime General Theory (PGT)**, which I've developed. The use of modular groups transforms computing of $2^p - 1$ into an arithmetic progressions of a single residue to a specially constructed modulus, whose computational form is super simple and easy to compute in software. The source and output of working code is provided to show actual results.

Thus, a many times more efficient|faster system than currently used by GIMPS can be configured. It could not only find new M_p values, but also the large primes between the consecutive M_p that are also congruent prime residues to the **MPG** in use.

Mersenne Primes Search

Mersenne Primes have the form: $M_p = 2^p - 1$ for p prime. They are very rare, and GIMPS (Great Internet Mersenne Prime Search) was created to find and record them. As of October 12, 2024, 52 values for p have been verified that make M_p a prime. Here is their list: 2, 3, 5, 7, 13, 17, 19, 31, 61, 89, 107, 127, 521, 607, 1279, 2203, 2281, 3217, 4253, 4423, 9689, 9941, 11213, 19937, 21701, 23209, 44497, 86243, 110503, 132049, 216091, 756839, 859433, 1257787, 1398269, 2976221, 3021377, 6972593, 13466917, 20996011, 24036583, 25964951, 30402457, 32582657, 37156667, 42643801, 43112609, 57885161, 74207281, 77232917, 82589933, 136279841. (sequences A000043 in the OEIS).

The goal is to use modular groups to lower the M_p **prime number space (pns)**, to greatly increase the search efficiency and speed, by identifying only the residues for p that can make $2^p - 1$ be primes.

We know P4 and P6 have only two residues, and it's easy to show Mersenne Primes exist only along one for each, and have generator forms: $P6 = 6k + 1$ | $P4 = 4k + 3$. Here are their proofs.

Proof: $2^p - 1 = 6k + 1$ for odd $p > 2$

$$\begin{aligned}2^p - 1 &= 6k + 1 \\2^p - 2 &= 6k \\2(2^{p-1} - 1) &= 6k \\2^{p-1} - 1 &= 3k\end{aligned}$$

Proof: $2^p - 1 \neq 6k + 5$

$$\begin{aligned}2^p - 1 &= 6k + 5 \\2^p - 6 &= 6k \\2^{p-1} - 3 &\neq 3k\end{aligned}$$

Since $2^{p-1} - 1$ and $2^{p-1} - 3$ are odd $3k$ must be odd, so k must be odd. Setting $k = 2a \pm 1$ in each case, and continuing the regression for both, will show for $6k+5$ either $2^{p-x} - 3 = 3a$ or $2^{p-x} = 3a$ at each stage. For $2^{p-x} = 3a$ the left side is even (a power of 2), thus not divisible by 3, and $2^{p-x} - 3 = 3a \Rightarrow 2^{p-x} = 3(a + 1)$, and again 2^{p-x} is not divisible by 3, thus $6k + 5$ cannot contain Mersenne Primes.

Using this same procedure we can prove Mersenne Primes for P4 have the form $4k + 3$.

Proof: $2^p - 1 = 4k + 3$

$$\begin{aligned}2^p - 1 &= 4k + 3 \\2^p &= 4(k + 1) \\2^{p-2} &= k + 1 \\2^{p-2} - 1 &= k\end{aligned}$$

Proof: $2^p - 1 \neq 4k + 1$

$$\begin{aligned}2^p - 1 &= 4k + 1 \\2^p &= 2(2k + 1) \\2^{p-1} &= 2k + 1 \\2^{p-1} - 1 &\neq 2k\end{aligned}$$

Thus for $M_p = 2^p - 1$, the prime values 2, 3, 5, 7 correspond to the k values 0, 1, 7, 31 for $4k + 3$.

It can also be shown all the k values for each generator are odd numbers which follow a specific sequence, and additionally, not only must p be prime but each prime must have form $p = 2n + 3$.

P4 creates a Mersenne Prime number space to 1/4 of all integers, and P6 reduces it even further to 1/6. But we can do even better by combining them to create a larger generator with a smaller M_p pns.

Since the M_p reside along a single retrack for P4 and P6, we can combine them. The primes will then exist when their modular cycles intersect, i.e. when the modulus is $4 \cdot 6 = 24$. There are 8 residues for $\phi(24)$, so $P24 = 24k + \{1, 5, 7, 11, 13, 17, 19, 23\}$, which generates all the primes > 3 . We want only one residue to generate the M_p , i.e. $2^p - 1 = 24k + r$ for only one r in $\{1, 5, 7, 11, 13, 17, 19, 23\}$. To find it, we numerate the P4|P6 sequences up to 24 and identify where their values intersect (are equal).

$$\begin{array}{rcccccc} k: & 0 & 1 & 2 & 3 & 4 & 5 \\ 4k + 3: & 3 & \underline{7} & 11 & 15 & \underline{19} & 23 \end{array}$$

$$\begin{array}{rcccccc} k: & 0 & 1 & 2 & 3 & 4 \\ 6k + 1: & 1 & \underline{7} & 13 & \underline{19} & 25 \end{array}$$

We thus now have two possibilities, $24k + 7 \mid 24k + 19$, and use the previous procedure to identify it.

Proof: $2^p - 1 = 24k + 7$ for $p > 3$

$$\begin{aligned} 2^p - 1 &= 24k + 7 \\ 2^p &= 8(3k + 1) \\ 2^{p-3} &= 3k + 1 \\ 2^{p-3} - 1 &= 3k \end{aligned}$$

Proof: $2^p - 1 \neq 24k + 19$

$$\begin{aligned} 2^p - 1 &= 24k + 19 \\ 2^p &= 24k + 20 \\ 2^p &= 4(6k + 5) \\ 2^{p-2} - 5 &\neq 6k \end{aligned}$$

Because $2^{p-2} - 5$ is odd, and $6k$ even, they can't equate, thus r must be 7, therefore $M_p = P24 = 24k + 7$.

The M_p *pns* has now been reduced from 1/4, to 1/6, to now 1/24 (4.167%) of all integers. It's also now possible to construct a family of progressively larger generators based on a general embedded pattern.

By observation (and some intuition), we see P6 and P24 can be rewritten to have a general M_p form:

$$P6 = 6k + 1 = (2^1 \cdot 3)k + 1 \quad P24 = 24k + 7 = (2^3 \cdot 3)k + 7 \Rightarrow M_p = (2^m \cdot 3)k + (2^m - 1) \text{ for } m \text{ odd}$$

For P6, $m = 1$ and $r = 2^1 - 1 = 1$, and for P24, $m = 3$ and $r = 2^3 - 1 = 7$.

It can be shown for m odd, a family of declining number space generators are constructed with form:

$$M_p = 2^p - 1 = (2^m \cdot 3)k + (2^m - 1), \text{ } m \text{ odd } (1, 3, 5, 7, \dots) \quad (1)$$

They are used in the following manner. Once you find a M_p exponent prime we can construct the next larger M_p generator to find the next one, reducing the number space to $1/(2^m \cdot 3) \%$ of all integers.

Use P4 $M_p = 4k + 3$ to find $M_2 = 3$ ($p = 2; k = 0$)
 Use P6, $m = 1, M_p = 6k + 1$ to find $M_3 = 7$, ($p = 3; k = 1$)
 Use P24, $m = 3, M_p = 24k + 7$ to find $M_5 = 31$, ($p = 5; k = 1$)
 Use P24, $m = 3, M_p = 24k + 7$ to find $M_7 = 127$, ($p = 7; k = 5$)
 Use P96, $m = 5, M_p = 96k + 31$ to find $M_{13} = 8191$, ($p = 13; k = 85$)
 Use P384, $m = 7, M_p = 384k + 127$ to find $M_{17} = 131071$, ($p = 17; k = 341$), and so on.

Again, each M_p generator can find all the M_p primes past their base residues. Each larger generator greatly reduces the number of M_p candidates values to search through by successively squeezing them into smaller number spaces, subsequently increasing their search efficiency and speed.

New Mersenne Primes Search

GIMPS is a distributed network of volunteers who are given different primes to test|recheck if they create an M_p prime. It must first identify ALL the primes within a range to test (an involved process by itself) before it distributes them. If a candidate M_p is found, it's value is rechecked multiple times to verify it. These valid p values become rarer (if they exist) as the primes density naturally decreases, thus a lot of time and work has to first be expended before any actual M_p value testing is even done.

A more efficient process uses **Mersenne Prime Generators (MPG)** to structurally generate them. An **MPG Search (MPGS)** eliminates the need to identify any primes, etc. Now, each distributed node uses the same **MPG** form, and just receives a unique k value to generate a candidate M_p prime. They then perform primality tests on their computed values by checking if its binary form is a string of '1' bits, and communicates back T|F. If true, the k value can be redistributed for rechecking, and if verified, then $p = \log_2(M_p + 1)$, and its binary form is a string of p consecutive '1' bits, e.g. $M_3 = 7 = 0b111$.

Let's illustrate the search using $MPG = 24k + 7$, where the modulus is 24 and its residue $M_3 = 7$. This **MPG** (by construction) will generate all the M_p for $p \geq 3$ (odd primes). In the table below we see the k values that generate the first 8 M_p values. It quickly finds the first 4 up to M_{13} , then the k values start to significantly increase to find the next 4. But we can optimize the generators (and their computations).

	$2^p - 1 = 24k + 7$							
k	0	1	5	341	5 461	21 845	89 478 485	96 076 792 050 570 581
M_p	$M_3=7$	$M_5=31$	$M_7=127$	$M_{13}=8191$	$M_{17}=131071$	$M_{19}=524287$	$M_{31}=2147483647$	$M_{61}=2305843009213693951$

To optimally implement the **MPG** they can be transformed to these optimal computational forms:

$$MPG = (2^p \cdot 3)k + (2^p - 1) = 2^p (3k + 1) - 1 = ((3k + 1) \ll p) - 1 \quad (2)$$

Thus, a known M_p value becomes the residue of an arithmetic progression that will numerate every larger M_p value. Each larger **MPG** then lowers the pns to find the next M_p , using smaller k values. And by eliminating the need to first directly identify (find|confirm) any primes, this frees up all that computing resource overhead to be applied to just computing k value candidate M_p values.

Therefore, to find the next larger Mersenne Prime to the current largest $M_{136279841}$ we create the **MPG**:

$$MPG = (2^{136279841} \cdot 3)k + M_{136279841} = ((3k + 1) \ll 136279841) - 1 \quad (3)$$

If another M_p exists, it will be a congruent residue of $M_{136279841}$ to the modulus $(2^{136279841} \cdot 3)$, which is a humongously large modular group size, with an infinitesimally small pns . Even so, a small percentage of infinity is still infinity.

Shown next are the progression of **MPG**, with their decreasing pns , and their k values that correspond to the first few M_p values. As an example, for $M_{31} = 2^{31} - 1 = 2,147,483,647$, the smaller **MPG** k values mean fewer resgroup pcs need checking for the same M_p , thus the number of M_p candidate values are structurally greatly reduced as the **MPG** moduli increase, to significantly reduce search times.

	$2^p - 1 = 24k + 7 \mid pns = 4.167 \%$							
k	0	1	5	341	5 461	21 845	89 478 485	96 076 792 050 570 581
M _p	M ₃	M ₅	M ₇	M ₁₃	M ₁₇	M ₁₉	M ₃₁	M ₆₁

	$2^p - 1 = 96k + 31 \mid pns = 1.042 \%$							
k		0	1	85	1 365	5 461	22 369 621	24 019 198 012 642 645
M _p		M ₅	M ₇	M ₁₃	M ₁₇	M ₁₉	M ₃₁	M ₆₁

	$2^p - 1 = 384k + 127 \mid pns = 0.26 \%$							
k			0	21	341	1 365	5 592 405	6 004 799 503 160 661
M _p			M ₇	M ₁₃	M ₁₇	M ₁₉	M ₃₁	M ₆₁

	$2^p - 1 = 24576k + 8191 \mid pns = 0.004 \%$							
k				0	5	21	87 381	93 824 992 236 885
M _p				M ₁₃	M ₁₇	M ₁₉	M ₃₁	M ₆₁

	$2^p - 1 = 393216k + 131071 \mid pns = 0.00025 \%$							
k					0	1	5 461	5 864 062 014 805
M _p					M ₁₇	M ₁₉	M ₃₁	M ₆₁

	$2^p - 1 = 1572864k + 524287 \mid pns = 0.0000636 \%$							
k						0	1 365	1 466 015 503 701
M _p						M ₁₉	M ₃₁	M ₆₁

	$2^p - 1 = 6442450944k + 2147483647 \mid pns = 0.0000000155 \%$							
k							0	357 913 941
M _p							M ₃₁	M ₆₁

Here I express the *MPG* in their standard modular group form, to show that the M_p values become the residues of each new *MPG*, and show what their modulus values are, which determine their *pns* values. We also see the growing spread between the k values corresponding to the M_p. Next I'll code the arithmetic structure of these *MPG* forms and perform a search with them.

MPGS Code

Below is Crystal code to search for the M_p . It starts with any known M_p prime p , and generates all the primes along its *MPG* residue track. Once a prime value is found it checks if its an M_p by checking if all it's bits are '1's. If so, it computes p , displays $[p, k, M_p]$, then starts searching with p for the next M_p .

```
# Single core example to do an Optimal Mersenne Primes Search using Modular Groups
# crystal build --release --mcpu=native mpgs.cr
require "big"
require "primes-utils"      # provides 'prime?'

def mpg(p, k) mp = ((k*3 + 1) << p) - 1 end
def mp_bits_check?(n) while (n >>= 1) > 0; return false if n.even? end; true end

p = ARGV.size > 0 ? (ARGV[0].to_u64 underscore: true) : 3u64
k = 1.to_big_i
tm = Time.instant
puts "Starting search for next Mp with known p = #{p}"

loop do
  mp = mpg(p, k)
  if mp.prime? && mp_bits_check?(mp)
    p = (Math.log(mp + 1)/Math.log(2)).to_u64
    pp [p, k, mp]
    puts "Elapsed time = #{Time.instant - tm}"
    puts "\nStarting search for next Mp with known p = #{p}\n"
    k = 1.to_big_i
    tm = Time.instant
  else
    k += 2
  end
end
```

Here is sample output starting from the known M_p prime shown.

```
$ ./mpgs 19
Starting search for next Mp with known p = 19
[31, 1365, 2147483647]
Elapsed time = 00:00:00.000820068

Starting search for next Mp with known p = 31
[61, 357913941, 2305843009213693951]
Elapsed time = 00:01:48.605697428

Starting search for next Mp with known p = 61
[89, 89478485, 618970019642690137449562111]
Elapsed time = 00:00:52.220594252

Starting search for next Mp with known p = 89
[107, 87381, 162259276829213363391578010288127]
Elapsed time = 00:00:00.059908606
```

This code directly implements the arithmetic expressions to compute the M_p . But by understanding the intent of the algorithm, it can be coded to **do what the math means**, instead of how its arithmetically shown to do it. This will simplify the code, and increase its speed by orders of magnitude.

As previously stated, $M_p = 2^p - 1$ has a binary representation 0b111..111, a p length string of '1' bits. Therefore, the expression $((3k + 1) \ll p) - 1$ has to produce similar values. This can only occur when $(3k + 1) = 2^m$, where m are even exponent powers of 2. This table shows valid k values for them.

$3k + 1$						
k	1	5	21	1 365	87 381	349 525
2^m	$4 = 2^2$	$16 = 2^4$	$64 = 2^8$	$4096 = 2^{12}$	$262144 = 2^{18}$	$1048576 = 2^{20}$

Thus to compute the candidate M_p , we can replace $(3k + 1)$ with even powers of 2. The expression now becomes $(2^m \ll p) - 1$, and is optimally computed as $(1 \ll (p + m)) - 1$, equivalent to $2^{p+m} - 1$. Since these values produce only binary strings of '1' bits, the code no longer needs to do bit checks on them, and m starts at 2 and is incremented by 2, thus eliminating the time to generate the k values. The code now becomes simpler, and orders of magnitude faster (but now only generates new M_p values).

```
# Single core example to do an Optimal Mersenne Primes Search using Modular Groups
# crystal build --release --mcpu=native mpgs.cr
require "big"
require "primes-utils"      # provides 'prime?'

def mpg(p, m) mp = (1.to_big_i << (p + m)) - 1 end

p = ARGV.size > 0 ? (ARGV[0].to_u64 underscore: true) : 3u64
m = 2u64
tm = Time.instant
puts "Starting search for next Mp with known p = #{p}"

loop do
  mp = mpg(p, m)
  if mp.prime?
    p = (Math.log(mp + 1)/Math.log(2)).to_u64
    pp [p, m, mp]
    puts "Elapsed time = #{Time.instant - tm}"
    puts "\nStarting search for next Mp with known p = #{p}\n"
    m = 2u64
    tm = Time.instant
  else
    m += 2
  end
end
end
```

These outputs show the orders of magnitude performance increase using the optimized math code.

```
$ ./mpgs 107
Starting search for next Mp with known p = 127
[521, 394,
68647976601306097149819007990813932172694353001433054093944634591855431833976560521225596406
61454554977296311391480858037121987999716643812574028291115057151]
Elapsed time = 00:00:00.002608888

Starting search for next Mp with known p = 521
[607, 86,
53113799281676709868958820655246862732959311772703192319944413820040355986085224273916250226
5229285668889329486246501015346579337652707239409519978766587351943831270835393219031728127]
Elapsed time = 00:00:00.002203116
```

This code, run on my laptop, was able to generate $M_{107} \rightarrow M_{127}$, then $M_{127} \rightarrow M_{521}$ in $< 3\text{ms}$ for each. The process stopped because of limitations of my system to perform the large number arithmetic to continue going from $M_{521} \rightarrow M_{607}$ and on. But given appropriate soft|hardware resources, we are no longer time bound to sequentially generate all the known M_p values, and beyond. Thus an individual, with the sufficient resources, can now engage in their own Mersenne Primes Search project, with the knowledge provided here.

Conclusion

All primes are residues of some modular groups, thus they can be characterized and understood from the properties and structure of modular groups. What's shown is Mersenne Primes of form $2^p - 1$, have nice unique modular representations, which can be used to create special *Mersenne Prime Generators*. The inputs to these generators are known primes p for some M_p value, which will generate the next M_p value, whose p value can be used to find the next one, and so on.

This process is much less resource intense than currently used by GIMPS. No list of primes is needed, requiring constant work to find, verify, and store them, before they can be used to run tests with. And as the M_p are very rare, most of the work to find all these primes ultimately lead to negative outcomes.

Thus, the process presented here uses far fewer computational and hardware resources, and will always find a new M_p when starting from a known one. Thus using the p value for the currently known largest M_p , it will find the next one (if it exists) in orders of magnitude less time than required by GIMPS.