On inversion modulo pseudo-Mersenne primes

Michael Scott

Cryptographic Researcher
MIRACL Labs
mscott@indigo.ie

Abstract. It is well established that the method of choice for implementing a side-channel secure modular inversion, is to use Fermat's little theorem. So $1/x = x^{p-2} \bmod p$. This can be calculated using any square-and-multiply method safe in the knowledge that no branching or indexing with potentially secret data (such as x) will be required. However in the case where the modulus p is a pseudo-Mersenne, or Mersenne, prime of the form $p = 2^n - c$, where c is small, this process can be optimized to greatly reduce the number of multiplications required. Unfortunately an optimal solution must it appears be tailored specifically depending on n and c. What appears to be missing from the literature is a near-optimal heuristic method that works well in all cases.
Keywords: Elliptic Curves

1 Introduction

In elliptic curve cryptography (ECC), a pseudo-Mersenne modulus is often proposed, as it introduces no known weaknesses, and allows a much faster modular reduction algorithm, independent of the underlying computer architecture. Less appreciated is the fact that modular inversions can also benefit from such a choice. For maximum efficiency projective coordinates are most often used for ECC, which greatly diminishes the significance of the cost of modular inversion. Nevertheless in a paper that popularised such moduli, Bernstein [1] pointed out that modular inversion still absorbed 7% of the time for a curve computation, and in a recent paper Nath and Sarkar [12] point out that as a consequence of the vigorous optimization of the other aspects of implementation, this may rise as high as 9%.

Unfortunately working out the optimal strategy in the general case where $p = 2^n - c$ is not that simple, as it depends on both n and c. Since the binary expansion of such a p mostly consists of 1 bits, a simple square and multiply algorithm will be particularly inefficient, requiring nearly as many multiplications as squarings. In fact it is impossible to avoid $n-1$ squarings. So any attempt at optimisation will focus on reducing the number of multiplications.

To get an idea of what is possible, let us deconstruct Bernsteins approach for his prime $2^{255} - 19$, as described by Bos [5]. The value after the # indicates the exponent of x at that stage in the calculation. In this case the inverse will be calculated as $1/x = x^{255} - 21$

$$
\begin{aligned}
x_2 &\leftarrow x^2 && \# \ 2\\
x_4 &\leftarrow x_2^2 && \# \ 4\\
x_8 &\leftarrow x_4^2 && \# \ 8\\
x_9 &\leftarrow x.x_8 && \# \ 9\\
x_{11} &\leftarrow x_2.x_9 && \# \ 11\\
x_{22} &\leftarrow x_{11}^2 && \# \ 22\\
t_0 &\leftarrow x_9.x_{22} && \# \ 2^5-1\\
\\
t_1 &\leftarrow t_0^{2^5}.t_0 && \# \ 2^{10}-1\\
t_2 &\leftarrow t_1^{2^{10}}.t_1 && \# \ 2^{20}-1\\
t_3 &\leftarrow t_2^{2^{20}}.t_2 && \# \ 2^{40}-1\\
t_4 &\leftarrow t_3^{2^{10}}.t_2 && \# \ 2^{50}-1\\
t_5 &\leftarrow t_4^{2^{50}}.t_4 && \# \ 2^{100}-1\\
t_6 &\leftarrow t_5^{2^{100}}.t_5 && \# \ 2^{200}-1\\
t_7 &\leftarrow t_6^{2^{50}}.t_4 && \# \ 2^{250}-1\\
\\
x &\leftarrow t_6^{2^5}.x_{11} && \# \ 2^{255}-21
\end{aligned}
$$

On closer examination the process consists of 3 phases. It makes extensive use of the identity

$$x^{2^{n+m}-1} = (x^{2^n-1})^{2^m} x^{2^m-1} \tag{1}$$

By the end of the second phase we have calculated $x^{2^{250}-1}$. Note that $c+2 = 21$ and $2^5 - 21 = 11$. The third phase then calculates $x^{2^{255}-21} = (x^{2^{250}-1})^{2^5}.x^{2^5-21}$. It is easily confirmed that the whole process requires 254 squarings and just 11 multiplications.

Clearly the final phase requires only one extra multiplication. In the first phase the important work is the calculation of $k = x^{11}$ and $x^{31} = x^{2^5-1}$. The former to provide the "key" value needed by the final phase, and the latter is used to kick-start the second phase with some value of the form x^{2^m-1}. Therefore there are two addition chain calculations involved, the first to determine x^{31}, with the constraint that the chain pass through the value of x^{11}. The second applies to the second phase where the point is to calculate in as few steps as possible $x^{2^{250}-1}$ using identity (1) above. The first chain requires a multiplication for each addition step in the chain (but not for a doubling). The second addition chain requires a multiplication for each step, so the shortest possible chain is desired.

To generalise this approach, first choose a value w such that $2^w > c+2$. Calculate the key $k = x^{2^w-c-2}$ and x^{2^w-1} in phase 1. Raise this value up to $x^{2^{n-w}-1}$ in phase 2, and finally calculate the inverse of x as x^{2^n-c-2} in phase 3. Clearly what makes this process rather awkward is the involvement of addition chains. As is well known calculating shortest addition chains is an NP-complete problem, and therefore not suitable to be calculated on-the-fly. So it appears that an optimal solution must be tailored by hand to each pseudo-Mersenne prime of interest.

The alternative is to come up with a heuristic approach which can quickly produce a good (if sub-optimal) solution given only n and c. That is the contribution of this paper.

2 A heuristic approach

Choose w to be the smallest number such that $2^w > c+2$. Next calculate and store powers of x using this fixed addition chain of exponents

$$[1\ 2\ 3\ 6\ 12\ 15\ 30\ 60\ 120\ 240\ 255]$$

This will require three multiplications and seven squarings. Next evaluate the key value $k = x^{2^w-c-2}$ by multiplying together appropriate powers, which will probably require a few more multiplications. It is easily confirmed that this is possible for any values of $2^w-c-2 < 745$, which will cover most cases of interest. Note that these stored values include all of $x, x^{2^2-1}, x^{2^4-1}, x^{2^8-1}$. Extract these values, and use them to initialise another addition chain in the exponents, where the entry i represents the power x^{2^i-1}.

$$[1\ 2\ 4\ 8\ 16\ ...\ 2^m\ ...\ n-w]$$

which is just the powers of 2 while $n-w < 2^m$. Finally complete the addition chain using a simple binary method. This chain dictates how to use the identity (1) to ramp up x^{2^8-1} to $x^{2^{n-w}-1}$. Finally calculate the result by squaring this final value w times, and multiplying it by the key k.

For the modulus $2^{255}-19$, $w=5$ and the key value will be $k = x^{11} = x^2.x^3.x^6$, which would require two extra multiplications. The addition chain in the exponents will be

$$[1\ 2\ 4\ 8\ 16\ 32\ 64\ 128\ 192\ 224\ 240\ 248\ 250]$$

which will require 9 further multiplications (recall that the first 4 entries are already available), plus one for phase 3. The total number of multiplications is 15, somewhat inferior to the optimal value of 11.

When it comes to implementing this method, note that the same array used to store the powers of x in phase 1, can be re-used in phase 2. Specifically the 11 element array used in phase 1 can be re-used in phase 2 for values of n less than 2048. Therefore the method as described will work for all primes of the form $2^n - c$ for $n < 2048$ and $c < 1024$. See algorithm 1.

3 Performance

To get an idea of just how suboptimal this general approach will be, we compare it with some implementations that are already "out there" in the wild, in existing

Algorithm 1 Modular inversion with respect to a pseudo-Mersenne prime

INPUT: An element $x \in \mathbb{F}_p$, n and c, where prime $p = 2^n - c$, $n < 2048$, $c < 1024$

INPUT: An array a=[1,2,3,6,12,15,30,60,120,240,255]

OUTPUT: $1/x \bmod p$

```
function FPINV(x, n, c)
    h[0] ← x                                          ▷ Phase 1
    h[1] ← x^2
    h[2] ← x.h[1]
    h[3] ← h[2]^2
    h[4] ← h[3]^2
    h[5] ← h[4].h[2]
    h[6] ← h[5]^2
    h[7] ← h[6]^2
    h[8] ← h[7]^2
    h[9] ← h[8]^2
    h[10] ← h[9].h[5]
    b ← 0
    w ← 1
    while w < c + 2 do
        w ← 2w
        b ← b + 1
    j ← w − c − 2
    if j ≠ 0 then
        i ← 10
        while a[i] > j do
            i ← i − 1
        k ← h[i]                                      ▷ Calculate Key
        j ← j − a[i]
    while j ≠ 0 do
        i ← i − 1
        if j ≥ a[i] then
            k ← k.h[i]
            j ← j − a[i]
                                                      ▷ Phase 2
    h[1] ← h[2]                                       ▷ Re-use the array
    h[2] ← h[5]
    h[3] ← h[10]
    j ← 3
    m ← 8
    n ← n − b
    while 2m < n do                                   ▷ Double up
        t ← h[j]
        j ← j + 1
        for i ← 0; i < m; i ← i + 1 do
            t ← t^2
        h[j] ← t.h[j − 1]
        m ← 2m
    l ← n − m
    r ← h[j]
    while l ≠ 0 do                                    ▷ Complete addition chain
        m ← m/2
        j ← j − 1
        if l ≥ m then
            l ← l − m
            t ← r
            for i ← 0; i < m; i ← i + 1 do
                t ← t^2
            r ← t.h[j]
                                                      ▷ Phase 3
    for i ← 0; i < b; i ← i + 1 do
        r ← r^2
    if w − c − 2 ≠ 0 then
        r ← r.k
    return r
```

code and libraries. Specifically we looked at the code associated with references [1], [6], [7], [12] and [13] in October 2018. In Table 1 we show the number of multiplications as implemented, and as required by the method described here. In cases where the number of squarings exceeds $n-1$, these are recorded as well. As expected our method is suboptimal, but not by much. In fact it may be considered as better than expected. For example Bos et al. [6] found that they could always calculate the modular inversion using at most $1.11 \lceil \log_2(p) \rceil$ squarings and multiplications. With our much larger sample we find that it can be done using at most $1.09 \lceil \log_2(p) \rceil$. As can be seen we also find that on occasion this method improves on a manually tailored solution.

Prime	As Implemented	This Method	Alternate Method
$2^{127}-1$	10 [12]	12	12
$2^{221}-3$	12 [12]	12	13
$2^{222}-117$	12 [12]	14	14
$2^{251}-9$	14 [12]	15	15
$2^{255}-19$	11 [1]	15	15
$2^{256}-189$	21(260) [6]	14	14
$2^{266}-3$	12 [12]	12	13
$2^{336}-3$	13 [13]	13	14
$2^{382}-105$	14 [12]	16	17
$2^{383}-187$	15 [12]	17	18
$2^{384}-317$	15 [6]	18	18
$2^{414}-17$	14 [12]	14	14
$2^{511}-187$	15 [12]	18	19
$2^{512}-569$	16(512) [6], 18 [12]	19	20
$2^{521}-1$	13 [7]	13	13
$2^{607}-1$	14 [12]	15	15
$2^{751}-165$	19 [12]	19	19
$2^{832}-143$	16 [12]	17	18
$2^{896}-213$	18 [12]	18	18
$2^{960}-167$	19 [12]	17	19
$2^{1024}-105$	20 [12]	18	19
$2^{1088}-89$	16 [12]	17	17

Table 1. Number of Multiplications (Squarings)

4 A Generalisation

Not all Mersenne-like primes that have been suggested for use in the context of ECC are of the simple form considered above. For example there are the generalised Mersenne primes [14] and various hybrid forms. Therefore it is natural to ask if the general purpose scheme described above can be easily extended to include more of these.

Here we consider prime moduli of the form $2^m - 2^n - c$, where $2n \leq m$ and c is small and positive. This covers at least two cases likely to be of particular interest, Hamburg's recently standardised Goldilocks curve [9] with a modulus of $2^{448} - 2^{224} - 1$, and the NIST standard curve `secp256k1`, as used in Bitcoin with its modulus of $2^{256} - 2^{32} - 977$. Observe that

$$x^{2^a - 2^b - c} = x^{(2^{a-b-1}-1)^{2^{b+1}}} x^{2^b - c}$$

From here the strategy is straightforward. Calculate the second term just as described above, and then extend the addition chain as necessary to calculate the larger first term. We omit the details.

Prime	As Implemented	This Method
$2^{448} - 2^{224} - 1$	13 [9]	15
$2^{256} - 2^{32} - 977$	15 [12]	17

Table 2. Number of Multiplications

5 Two birds, one stone

The same idea can be used to efficiently calculate modular square roots, as often required in ECC implementation for point decompression [8]. If the prime $p = 3 \bmod 4$, then the square root of a quadratic residue x can be calculated from $y = x^{(p-3)/4}$ as $\sqrt{x} = xy \bmod p$, and if $p = 5 \bmod 8$, the square root can be calculated from $y = x^{(p-5)/8}$, with a small amount of extra work – see algorithm 3.37 in chapter 3 of [11] for details. By modifying our algorithm in an obvious way to calculate these y values instead, and then re-using the same function, the inverse $x^{p-2} \bmod p$ can be found in the former case as xy^4, and in the latter case as x^3y^8. See [4] and the implementation associated with [9] for an example of the deployment of this idea.

Next we generalise this idea. First we categorise the prime moduli according to the value of e, where e is the maximum integer such that $2^e | p - 1$. If as is commonly the case the modulus is $3 \bmod 4$, then $e = 1$ and if the modulus is $5 \bmod 8$ then $e = 2$. Larger values of e might also occur for the case where the modulus is $1 \bmod 8$.

Now in the general case both modular inverses and tests for quadratic residuosity can start with the calculation of the *progenitor* value y

$$y = x^{(p-2^e-1)/2^{e+1}} \bmod p$$

From here the modular inverse can be calculated as

$$1/x = x^{2^e-1}.y^{2^{e+1}} = x^{2^{e-1}-1}.(xy^4)^{2^{e-1}} \bmod p$$

Quadratic residuosity can be determined from $(xy^2)^{2^{e-1}}$. If an element x is a quadratic residue, then its square root can be calculated.

We acknowledge that using this more round-about method to calculate modular inverses and modular square roots may require a small amount of extra work – for the former case see the last column in table 2. And in certain cases modular square roots can be achieved significantly faster, as $\sqrt{x} = x^{(p+1)/4}$ if $c = 1$, and from $\sqrt{x} = x^{(p+3)/8}$ if $c = 3$, as clearly in these cases only squarings are required, with no multiplications at all.

When considering the calculation of square roots it is common in the literature to make assumptions about p, as the calculation is much simpler for the cases where $e = 1$ and $e = 2$. Here we eschew this attractive but specialised approach and instead use the more complex Tonelli-Shanks algorithm (algorithm 3.34 [11]), which is normally reserved only for the "difficult" $p = 1 \bmod 8$ case. This requires knowledge of a fixed quadratic non-residue d, which ideally should be small. As is well known in the case $e = 1$, then $d = -1$ is a good choice. For the case $e = 2$, then $d = 2$. However for $e > 2$ often no automatic value is available, and indeed no deterministic algorithm is known which can find one. In practice however the prime modulus is known in advance and a simple off-line search through the small primes will quickly find a suitable non-residue. Also needed will be a precomputed 2^e-th root of unity, that is $z = d^{(p-1)/2^e}$.

In the application of Tonelli-Shanks we again assume that the progenitor y is first calculated as above. Indeed this may already be available from a prior test for quadratic residuosity, as it would obviously be pointless to proceed to try and find the square root of a non-square. This is the case when performing elliptic curve point decompression for example. Then the algorithm to find the square root of x proceeds as follows (in a Pythoneque pseudo-code, where the `cmov` function moves the second parameter into its first parameter if the condition specified in its third parameter is true. Such a function is a staple of constant-time implementation).

```
s=y*x
t=s*y
for k in range(e,1,-1) :
    b=t
    for i in range(1,k-1) :
        b*=b
    cmov(s,s*z,b!=1)
    z*=z
    cmov(t,t*z,b!=1)
```

The square root will be the final value of s. Observe that in the case where $e = 1$ the for loop is not executed.

Next consider the cost of this algorithm in terms of e. The method above takes $2e - 1$ multiplications and $(e^2 - e)/2$ squarings. For larger values of e this is offset to a small extent, as for the same size of p the initial calculation of y will cost e less squarings. Overall the extra costs will not be excessive for moderate values of e.

This ability to calculate both inverses and square roots from the same initial computation of the progenitor y, leads immediately to the "inverse square root" trick [4], where

$$\sqrt{u/v} = u^2 . \frac{1}{u^3 v} . \sqrt{u^3 v}$$

This calculation, requiring just one modular exponentiation, is relevant if implementing the Elligator2 method for deterministic mapping to elliptic curve points [3], or if performing point decompression on Edwards curves [4].

In the past prime moduli of the form $p = 1 \bmod 8$ have been avoided due to their perceived difficulty, and often libraries did not offer support. However the extra cost and complexity is in fact not excessive for moderate values of e. This opens up more possibilities for elliptic curve moduli with an exploitable form for fast implementation, for example the primes $2^{255} - 31$ and $2^{383} - 31$. We observe that in the original paper introducing Curve25519 Bernstein [1] preferred $2^{255} - 19$ over $2^{255} - 31$ solely because "19 is smaller than 31". However in later work [2] the prime $2^{414} - 17$ is recommended as 17 is close to a power of 2, as also is 31. A suitable Weierstrass curve for $p = 2^{255} - 31$ of prime order (and with prime order twist) would be $y^2 = x^3 - 3x + 5313$.

6 Exploiting the Scholz-Brauer conjecture

Finally we re-consider our approach for primes of the particular generalised Mersenne form $2^n - 2^e + 1$, where $e > 0$. As can be easily observed in these cases the progenitor y becomes $y = x^{2^m - 1} \bmod p$ where $m = n - e - 1$. In this case the well-known Scholz-Brauer conjecture on addition chains applies (see [10], Chapter 4.6.3).

$$l(2^m - 1) \leq m - 1 + l(m)$$

where $l(m)$ is the length of the optimal addition chain for m. No counter-example to this conjecture has been found, and indeed in many cases the equality relationship is known to hold. Therefore we can take $m - 1 + l(m)$ to be very close to optimal. This suggests the following much simpler (and almost certainly superior) algorithm for calculating the progenitor y.

First find a simple shortest star chain for m, using tools freely available on the net. For example for $p = 2^{255} - 31$, then $m = 249$, and a suitable chain of length 10 would be

$$1, 2, 3, 6, 9, 15, 30, 60, 120, 240, 249$$

Now see algorithm 2, inspired by the proof of Theorem G in [10], Chapter 4.6.3. The performance comparison for the calculation of the progenitor value y with our original algorithm 1, is shown in table 3. For an extension of this method to a somewhat larger class of pseudo-Mersenne primes, see the Appendix.

Algorithm 2 Calculation of progenitor y for pseudo-Mersenne prime of the form $2^n - 2^e + 1$

INPUT: An element $x \in \mathbb{F}_p$, n and e, for prime $p = 2^n - 2^e + 1$
INPUT: An addition chain length L for $m = n - e - 1$, a=[1,2,...]
OUTPUT: $y = x^{2^m - 1} \bmod p$

```
function FPY(x)
    r ← x
    h[0] ← x
    f[1] ← 0
    for k ← 1; k ≤ L; k ← k + 1 do
        d ← a[k] − a[k − 1]
        for i ← 0; i < d; i ← i + 1 do
            r ← r^2
        r ← r.h[f[d]]
        h[k] ← r
        f[a[k]] ← k
    return r
```

Prime	Algorithm 1	Algorithm 2
$2^{127} - 1$	11	9
$2^{221} - 3$	11	10
$2^{255} - 31$	12	10
$2^{266} - 3$	11	11
$2^{336} - 3$	12	11
$2^{383} - 31$	13	11
$2^{495} - 31$	13	11
$2^{521} - 1$	12	12
$2^{607} - 1$	14	12

Table 3. Number of Multiplications

7 Conclusion

When implementing an easy-to-maintain general purpose cryptographic library, it helps to avoid duplication and special case implementation where possible, while still obtaining respectable performance. In the case of side-channel resistant modular inversion with respect to pseudo-Mersenne primes, the method of choice is to use Fermat's Little Theorem. Optimal performance seems to require a tailored solution for each modulus of interest, a process prone to error. Here we have first presented a slightly suboptimal algorithm which provides acceptable performance, but using a single function which works well for most cases likely to be of interest to cryptographers. Then we extend the idea to cover the determination of quadratic residuosity and the calculation of modular square roots, for primes of any form. Finally we present a much simpler method that often gives improved performance for a large subset of pseudo-Mersenne prime moduli.

One last observation: Basically the whole problem goes away if instead we were to use moduli of the form $2^n + c$ for positive values of c. Again such primes were also once considered by Bernstein [1]. And there are some nice such primes available, such as $2^{255} + 95$, $2^{263} + 9$, $2^{390} + 3$ and $2^{510} + 15$. One objection might be to the extra bit required in the representation, but in many cases, depending on how exactly field elements are represented, this might not be an issue.

References

1. D. Bernstein. Curve25519: New Diffie-Hellman speed records. In *PKC 2006*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer Berlin Heidelberg, 2006.
2. D. Bernstein, C. Chuengsatiansup, and T. Lange. Curve41417: Karatsuba revisited. In *Cryptographic Hardware and Embedded Systems – CHES 2014*, volume 8731 of *Lecture Notes in Computer Science*, pages 316–344. Springer Berlin Heidelberg, 2014.
3. D. Bernstein, M. Hamburg, M. Krasnova, and T. Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In *Proceedings of the 2013 ACM SIGSAC conference on computer and communications security*, pages 967–980, 2013.
4. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. Cryptology ePrint Archive, Report 2011/368, 2011. `http://eprint.iacr.org/2011/368`.
5. J. Bos. Constant time modular inversion. *Journal of Cryptographic Engineering*, 4(4):275–281, 2014.
6. J. Bos, C. Costello, P. Longa, and M. Naehrig. Selecting elliptic curves for cryptography: An efficiency and security analysis. Cryptology ePrint Archive, Report 2014/130, 2014. `http://eprint.iacr.org/2014/130`.
7. R. Granger and M. Scott. Faster ECC over $\mathbb{F}_{2^{521}-1}$. In *Public-Key Cryptography – PKC 2015*, volume 9020 of *Lecture Notes in Computer Science*, pages 539–553. Springer Berlin Heidelberg, 2015.
8. M. Hamburg. Fast and compact elliptic-curve cryptography. Cryptology ePrint Archive, Report 2012/309, 2012. `http://eprint.iacr.org/2012/309`.

9. M. Hamburg. Ed448-Goldilocks, a new elliptic curve. Cryptology ePrint Archive, Report 2015/625, 2015. `https://github.com/otrv4/little-ed448-Goldilocks`.
10. D. Knuth. *The Art of Computer programming, Volume 2: Seminumerical Algorithms, Third edition.* Addison-Wesley, 1997.
11. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of applied cryptography.* CRC Press, Boca Raton, Florida, 1996. URL: `http://cacr.math.uwaterloo.ca/hac`.
12. K. Nath and P. Sarkar. Efficient inversion in (pseudo-)Mersenne prime order fields. Cryptology ePrint Archive, Report 2018/985, 2018. `http://eprint.iacr.org/2018/985`.
13. M. Scott. Ed3363 (highfive) – an alternative elliptic curve. Cryptology ePrint Archive, Report 2015/991, 2015. `http://eprint.iacr.org/2015/991`.
14. J. Solinas. Generalized Mersenne numbers. Technical CORR-39, University of Waterloo, 1999.

Extending Scholz-Brauer method to a larger class of pseudo-Mersenne primes

Consider generalised pseudo-Mersenne primes of the form $2^n - 2^{d+e+1} + 2^{e+1} - 2^e + 1$, where $e > 0$ and $d \geq 0$. In this case the progenitor becomes $y = x^{2^m - 2^d} \bmod p$, which can be calculated as $y = x^{(2^{m-d}-1)2^d}$. Therefore the same method as described can be used, followed by d squarings, without requiring any more multiplications.

Now for primes $p = 2^n - c$, if $e = 1$ this method applies to values of $c = 2^{d+2} - 3$, for example $c = 1, 5$ and 13. If $e = 2$ the method applies for $c = 2^{d+3} - 5$, which includes $c = 3, 11, 27$. For $e = 3$, then $c = 2^{d+4} - 9$, which includes $c = 7, 23$. Therefore this simpler and faster method can be used for $c = 1, 3, 5, 7, 11, 13, 15, 23, 27, 29, 31...$