# ELLIPTIC CURVE FACTORIZATION AND MULTIPROCESSOR PROGRAMMING

CHRISTOPHER JEONG

Contents

1. Introduction 1
2. Cryptographic Motivations 1
3. A Brief Introduction to Multiprocessor Programming 2
4. Algorithm 2
5. Runtime Heuristics 4
6. Possible Improvements 5
7. References 5

## 1. Introduction

This paper discusses optimizations to Lenstra's Elliptic Curve Factorization Method (ECM) Algorithm. In particular, this paper will discuss multiprocessor programming and implementing the "stage 2" algorithm utilizing caching.

## 2. Cryptographic Motivations

Elliptic Curves serve as a cryptographic analogue to finite fields; problems in finite fields such as taking discrete logarithms, Diffie-Helman, and factoring have elliptic curve counterparts. Indeed, in this paper, we will discuss the latter problem and an algorithm to more efficiently factor elliptic curves.

Prior to any discussion regarding Lenstra's ECM Algorithm, it is critical to discuss Pollard's $p-1$ algorithm for factorization in finite fields. Pollard's $p-1$ is a finite field analogue for Lenstra ECM, which occurs on elliptic curves. See Figure 1 for pseudocode for Pollard $p-1$.

The problem that Pollard's $p-1$ algorithm is trying to solve is the problem of factoring $N = pq$, where $p$ and $q$ are prime. Consider the case where we have an integer $L$ such that $p-1|L$ and $q-l \nmid L$. By the definition of division, it follows that $i(p-1) = L, j(q-1)+k = L$ for some $i, j, k \in \mathbb{Z}, k \neq 0$. Now consider $a^L$. We can leverage Fermat's little theorem to yield two different results.

*Date*: May 2024.

```
def pollardPMinusOne(N : int) -> int:
    a = 2
    for j in range(2, bound):
        a = a^j mod N
        d = gcd(a - 1, N)
        if d > 1 and d < N:
            return d
    raise Exception("Failed to find a nontrivial factorization!")
```

Figure 1. Pollard's p - 1 algorithm

We will first discuss how such an integer $L$ is chosen; observe that $p-1|n!$ for some $n \in \mathbb{Z}$, and it is unlikely that $q-1|n!$. Thus, we compute $L$ in such a manner.

Consider some random integer $a$. We will first do some casework on $a, p$, and $q$. Consider the case where $p|a$ and $q|a$. If $p, q|a$, then $N|a$ since $p, q$ are coprime. If $N|a$, then $gcd(a, N) = N$, and we should pick another value of $a$. Now, without loss of generality, say $p|a$ and $q \nmid a$. If $p|a$ and $q \nmid a$, then we get that $gcd(a, N) = p$, and thus, we can compute $p$ easily. The same can be done for $q$. Thus, we can assume that $p \nmid a$ and $q \nmid a$.

Now consider $a^L$. We know that $a^L = a^{i(p-1)}$ by construction. If $a^L = a^{i(p-1)}$, then $a^L = (a^{p-1})^i$. By Fermat's little theorem, we know that $a^{p-1} \equiv 1 \mod p$. Thus, if $a^L = a^{i(p-1)}$, then $a^L \equiv 1^i \equiv 1 \mod p$. If $a^L \equiv 1^i \equiv 1 \mod p$, then $a^L - 1 \equiv 0 \mod p$, and it thus follows that $p$ must divide $a^L - 1$.

Let us consider $a^L$ once again. We know that $a^L = a^{j(q-1)+k}$ by construction. If $a^L = a^{j(q-1)+k}$, then $a^L = a^k a^{j(q-1)}$. If $a^L = a^k a^{j(q-1)}$, then $a^L = a^k (a^{q-1})^j$. By Fermat's little theorem, we know that $a^{q-1} \equiv 1 \mod q$. Thus, if $a^L = a^k (a^{q-1})^j$, then $a^L \equiv a^k \mod q$. If $a^L \equiv a^k \mod q$, then $a^L - 1 \equiv a^k - 1 \mod q$. Probabalistically, we know that $a^k$ is not congruent to $1 \mod q$; thus, it follows that $q \nmid a^L - 1$.

Now consider $gcd(a^L - 1, N)$. We know that $p|a^L - 1$, $q \nmid a^L - 1$, and that $p, q$ are the only nontrivial factors of $N$; thus, it follows that $gcd(a^L - 1, N) = p$. Earlier, we claimed that such an $L$ takes the form of $n!, n \in \mathbb{Z}$. This is exactly what the code in Figure 1 is doing: first it computes $a^2$, then $(a^2)^3 = a^{3!}$, then $(a^{3!})^4 = a^{4!}$, etc.

However, it is critical to note that Pollard's $p-1$ algorithm hinges entirely upon the fact that $p-1$ or $q-1$ factors into small primes; thus, one can choose $N = pq$ such that $p-1, q-1$ do not factor into relatively small primes, thus preventing Pollard's $p-1$ algorithm from working. As we will discuss later, Lenstra ECM fixes this problem.

## 3. A Brief Introduction to Multiprocessor Programming

As background for later discussions of concurrent programming, it is also crucial to discuss the notions of processes, threads, and multiprocessor computer systems. A process is an instance of a computer program. Each process has its own area of memory that it alone can modify.Processes also have one or many threads associated with them. A thread is similar to a process in that threads are also instances of computer programs. One area where threads and processes differ is that threads associated with a process share memory space. However, this difference will be trivial to our implementation.

For this implementation of ECM, we will be distributing the work between multiple processors. In other words, all different processes will have distinct memory areas for the most part. There will be a shared dictionary/hashmap kept in between process to store flags indicating each process's status. For the most part, however, the intricacies of multiprocessor programming need not be discussed; simply being aware that multiple instances of Lenstra ECM are running concurrently is sufficient.

## 4. Algorithm

Please refer to the attached .py file for more technical details of implementation. Due to space limitations, only the basic pseudocode of Lenstra ECM is included.

Section 2 discusses Pollard's $p-1$ algorithm; Lenstra ECM will be based heavily upon the same concepts but will rely on an elliptic curve group rather than a finite field. We will be working in the group $E(\mathbb{F}_p)$ consisting of points on the curve $E : Y^2 = X^3 + AX + B$, where $\mathbb{F}_p \cong \mathbb{Z}/p\mathbb{Z}$ is a finite field of order $p$, $A, B \in \mathbb{F}_p$ satisfying $4A^3 + 27B^2 \neq 0$. Suppose that we wish to factor $N = pq$, $p, q \in \mathbb{Z}$ are prime.

Consider the ring $\mathbb{Z}/N\mathbb{Z}$. We get that this ring is not a field because it is not an integral domain, i.e.

```
def lenstraECM(N : int) -> int:
    try:
        A, x, y = random(N)
        P = (x, y)
        for i in range(1, MAX_LENGTH):
            k = get_prime_power_near_bound(i, bound)
            P = k * P
    except InversionError as e:
        print("found a factor: " + str(e.factor))
```

FIGURE 2. Basic Lenstra ECM algorithm

$pq = 0, p, q \neq 0$. Now consider an elliptic curve $E : Y^2 = X^3 + AX + B$ over $\mathbb{Z}/N\mathbb{Z}$. It is critical to note that, because $\mathbb{Z}/N\mathbb{Z}$ is not a field, when performing operations such as addition and multiplication, calculating the inverses of $0, p, q$, and no other elements, will fail. The proof goes as follows:

We will first prove that $p, q$ have no inverses. Without loss of generality, consider $p$. Suppose, for the sake of contradiction, that $\exists p^{-1} \in \mathbb{Z}/N\mathbb{Z}$ such that $pp^{-1} = p^{-1}p = 1$. We know $pq = 0$, $p, q \neq 0$. It follows that $p^{-1}pq = 0$, further implying that $q = 0$. However, by assumption, $q \neq 0$, so we have a contradiction; thus, $p$ does not have an inverse.

We will now prove that all other elements are invertible. Consider some $g \in \mathbb{Z}/N\mathbb{Z}, g \neq 0, p, q$. Showing that $g$ is invertible is equivalent to $gcd(g, N) = 1$. We know that $N = pq$, where $p, q$ are prime. Thus, the only factors of $N$ are $1, p, q, pq$. We know that $g \neq p, q$. We also know that $g \in \mathbb{Z}/N\mathbb{Z}$; thus, $g < N$, implying that $g \neq pq$. Thus, $g$ shares no common factors with $N$ besides 1, and it follows that $gcd(g, N) = 1$. If $gcd(g, N) = 1$, then $g$ is invertible for all $g \in \mathbb{Z}/N\mathbb{Z}, g \neq 0, p, q$.

We have proven that inversion in $\mathbb{Z}/N\mathbb{Z}$ only fails when we invert $0, p$, or $q$; thus, if we fail to calculate an inverse, we know that we have attempted to invert one of the aforementioned elements. Now consider some $x \in \mathbb{Z}$. We will prove that $x$ cannot be inverted in $\mathbb{Z}/N\mathbb{Z}$ if $x$ is a multiple of $p$, or $q$. Let us work with $p$ without loss of generality. Suppose $x = np$ for some $n \in \mathbb{Z}$. We get that $gcd(x, N) = p$, implying that $x$ is not invertible in $\mathbb{Z}/N\mathbb{Z}$. The only factors of $N$ are $1, p, q, pq$; thus, for all $x$ that are not multiples of $p$ or $q$, $x$ is invertible.

So far we have proven that inversion fails if we attempt to invert a multiple of $p$ or $q$. We will now discuss which points in the algorithm inversion fails, implying that a factor has been found. This discussion will involve two stages: Stage 1 and Stage 2.

In order to discuss Stage 1 and Stage 2, we must introduce the concept of smoothness in numbers. A number is $B$-smooth if all of its prime factors are less than or equal to $n$. In Lenstra ECM, we provide two bounds: $B_1$ and $B_2$, each of which correspond to stage 1 and stage 2. For our implementation of the algorithm $B_1$ is chosen to be $11,000$ and $B_2$ is chosen to be $100 * B_1$. $B_1 = 11,000$ is in line with the standard value of a 25-digit number, but one can change such a boundary to accomodate larger numbers. We choose bounds primarily so that the algorithm knows when to stop looking for factors, i.e. it doesn't run indefinitely.

Stage 1 involves the repeated computation of $Q = kP$, where $k$ is the product of small prime powers. One may ask what happened to the $k!$ discussed with Pollard's $p - 1$ algorithm; since prime powers are sparser than all of the integers, we can reduce the number of elliptic curve multiplications occurring, thus reducing computational costs. Furthermore, we use powers of small primes in order to increase the number of factors of $k$, thus increasing the likelihood that an inversion will fail.

More specifically, we take a prime $p_0$, and then compute $p_0^{m_0}$, where $p_0^{m_0}$ is the largest power of $p_0$ that is not larger than $B_1$. We then compute $Q = p_0^{m_0}P$. Afterwards, we take another prime $p_1$, and then compute $p_1^{m_1}$, where $p_1^{m_1}$ is the largest power of $p_1$ that is not larger than $B_1$. We then compute $Q = p_1^{m_1}p_0^{m_0}P$. We keep going until a multiplication fails, which implies that we have found a factor.

It is critical to note that stage 1 works if $N$ is $B_1$-smooth, i.e. $N$'s prime factors are no greater than $B_1$. However, in the case that $N$ has a prime factor that in between $B_1$ and $B_2$, we must go to stage 2.

Stage 2 is similar to stage 1 in that it involves multiplication until failure. Consider the last $Q$ we computed in stage 1. At this point, $Q = p_0^{m_0}...p_t^{m_t}P$ for some primes $p_0, ..., p_t$ and nonzero natural numbers $m_0, ..., m_t$. We now hope to identify a factor $q$ of $N$ that satisfies $B_1 \leq q \leq B_2$; similar to Stage 1, we can continue to compute $Q = qP$ for primes $B_1 \leq q \leq B_2$, hoping for inversion to fail somewhere. However, computing primes that are similar to the magnitudes of $B_1$ and $B_2$ is expensive.

There appears to be two forms of step 2 of Lenstra ECM implemented across literature: one involves a Babystep-Giantstep algorithm that attempts to cover most primes in the range of $B_1$ and $B_2$, while the other simply relies on the precomputation of primes between $B_1$ and $B_2$. The Babystep-Giantstep algorithm appears to be designed for Elliptic curves with projective coordinates with group order divisible by 12 (and, when implemented with Weierstrauss Curves, did not appear to work!); thus, we will rely on the precomputation of primes between $B_1$ and $B_2$.

Lenstra ECM involves operations on many different elliptic curves. However, $B_1$ and $B_2$ remain the same; thus, we can just compute all primes up to $B_2$ once and reuse the same values over and over again. This is the strategy used in our corresponding implementation of Lenstra ECM and works much quicker than the Babystep-Giantstep algorithms used for Suyama's Parameterization of Montogomery Curves, i.e. specialized elliptic curves with group order divisible by 12 that are of the form $By^2z = x^3 + Ax^2z + xz^2 \mod N$.

## 5. Runtime Heuristics

It is critical to note that, for this multiprocessing implementation of Lenstra ECM, measuring runtime can be highly unpredictable. We use all available cores of a computer to perform computations on elliptic curves, i.e. all available computer resources are devoted to running our implementation. All test runs took place with a minimal number of background processes, but note that some inaccuracies may have been caused by background processes. Also note that this implementation tends to push computer capability: expect the computer fan to start running. For best performance, ensure that the fan is quiet and background processes are kept to a minimum. It is also important to note that Lenstra ECM is probabilistic; it may not return any answer and run indefinitely. Our current implementation of Lenstra ECM "recycles" processes in that should a process terminate with no valid return value, we simply spawn a new process in its place. This is where the advantages of Lenstra ECM over Pollard $p - 1$ appear; the group structure of the finite field cannot change while the group structure of the elliptic curve can change simply by picking new parameters.

We measured runtimes of Lenstra ECM utilizing python's time package and computed averages and standard deviations for the runtimes of the factorization of 64-bit, 72-bit, 80-bit, 88-bit, 96-bit, 104-bit, 112-bit, 128-bit, and 136-bit products of primes. Each trial except for 136-bits used 50 samples, while 136-bits used 10 samples. The mean runtime for factoring 64-bit products of primes was 0.2785 seconds, with standard deviation of 0.1508. The mean runtime for factoring 72-bit products of primes was 0.3646 seconds, with standard deviation of 0.2770 seconds. The mean runtime for factoring 80-bit products of primes was 0.7660 seconds, with standard deviation of 0.6502 seconds. The mean runtime for factoring 88-bit products of primes was 1.2587 seconds, with standard deviation of 1.1585 seconds. The mean runtime for factoring 96-bit products of primes was 2.3164 seconds, with standard deviation of 2.4529. The mean runtime for factoring 104-bit products of primes was 5.5312 seconds, with standard deviation of 4.7787 seconds. The mean runtime for factoring 112-bit products of primes was 13.2844 seconds, with standard deviation of 13.7933 seconds. The mean runtime for factoring 128-bit products of primes was 62.1788 seconds, with standard deviation of 61.3151. The mean runtime for factoring 136-bit products of primes was 184.0477 seconds, with

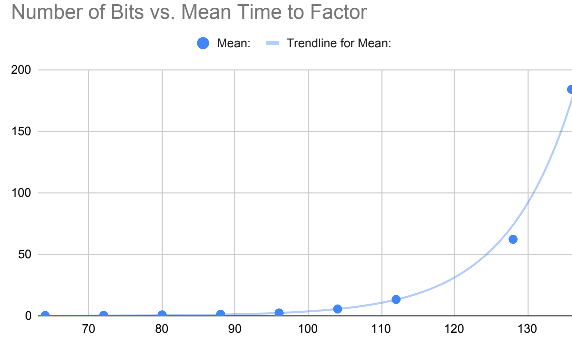


FIGURE 3. Plot of Number of Bits vs. Mean Time to Factor

standard deviation of 194.9749.

The results indicate that the time to factor increases exponentially as the number of bits increases. Indeed, the analysis of the runtime for this probabilistic algorithm results in a aggregate runtime of $O(e^{(\log p)(\log \log p)(1+O(1))})$, where $p$ is a factor of $N$. Such an analysis is beyond the scope of this course and will not be discussed. As noted from the very high standard deviations, this algorithm's runtime is highly unpredictable. Occasionally we will get lucky and a process will find a factor within a matter of seconds to multiple minutes.

## 6. Possible Improvements

Implementing the above algorithm in a compiled language like C or Rust rather than an interpreted language like python would improve both compilation time and runtime significantly. Although the discussion of such optimizations is best left to a dedicated compilers course, the most important difference between compiled and interpreted languages is that compiled languages are translated directly into machine code while interpreted languages are first translated into a compiled language and then into machine code; more translations results in slower compilation time as well as some loss of potential optimizations.

An algorithmic improvement that can be made is the use of projective coordinates and the use of Suyama Parameterized Montogomery curves rather than Weierstrauss curves. Using projective coordinates with Montogomery curves permits for additions and duplications without the computation of inverses, which are computationally expensive. Furthermore, Suyama Parameterized Montogomery curves allow us to control the group order of the curve so that it is divisible by 12. Such curves allow for a quicker method of elliptic point addition and multiplication known as Montgomery's ladder, which relies on differential addition and memoization to work and thus are faster than our implementation of Lenstra ECM at the cost of being expensive memory-wise.

## 7. References


- "Elliptic Curve Method." RieselPrime, www.rieselprime.de/ziki/Elliptic_curve_method. Accessed 6 May 2024.
- Washington, Lawrence C. Elliptic Curves: Number Theory and Cryptography. Chapman and Hall/CRC, 2008.
- Zimmermann, Paul, and Bruce Dodson. "20 years of ECM." Lecture Notes in Computer Science, 2006, pp. 525–542, https://doi.org/10.1007/11792086_37.