Pollard's p-1 and Lenstra's factoring algorithms

Anne-Sophie Charest

October 2, 2005

Abstract

This paper presents the result of my summer research on Lenstra's algorithm for factoring with elliptic curves. It first describes Pollard's p-1 algorithm, which is in a way the basis for Lenstra's algorithm. The theory behind both algorithms will be discussed, as well as the their detailed steps, their implementation and their efficiency. This paper is intended for people with little previous knowledge in group theory. Understanding of basic algebra, such as the ideas of gcd and modular arithmetic, is assumed.

Contents

Introduction

Number theory, although once renowned and loved for its lack of application to the real world, is now used in many ways in our everyday life. It's the basis of the the RSA cryptography system [8], used by millions of people everyday to exchange secret information, such as credit card numbers, over the Internet. This cryptosystem relies on the fact that it is much more easier to test for primality than to factor integers, which means that we can find two big primes and multiply them together without too much trouble, but we can not compute the unique factorisation into primes of the resulting composite in a reasonable amount of time. (For more detailed information on how the RSA cryptographic system and general concepts of cryptography, see [12].) However, the difficulty of factoring integers has not yet been proven, and this entire system would collapse if it were false and an efficient factoring algorithm were invented. This is what motivates in great part the search for fast factoring.

The idea of factoring an integer into primes is a very simple one. It can be performed properly by any elementary school student by dividing the integer to factor succesively by all integers smaller than its squareroot. This technique is straigthforward and always successful, but requires many operations. Inded, to find one factor of an integer n we might have to try up to $\sqrt{n}$ possible divisors. Thus, when the numbers to factorize get huge, such as in the cryptographic setting, this technique becomes rapidly unsuable. To this date, no computer is fast enough to factor large integers of about 300 digits in fewer than millions of years [3]. Hence, even if the speed of our computers would increase drastically, we would still rapidly run into numbers we couldn't factor in less than thousands of years.

So, the search for faster factorisation is really the search for new algorithms that factor integers in less operations. This report will present two algorithms that I have studied over the summer : Pollard's p-1 algorithm and Lenstra's algorithm using elliptic curves, the first one being the basis of the other one. Although they are not the fastest current algorithms to factor any integer, they can yield impressive results for factoring some particular kind of integers.

1 Pollard's p-1 algorithm

1.1 General idea of the algorithm

The $p-1$ algorithm was developped by J.M.Pollard in the 1970's [6] . The basic idea of the algorithm is to use some information about the order of an element of the group $\mathbb{Z}_p$ to find a factor p of N. The algorithm is based on the following theorem :

Theorem 1.1. *Fermat's Little Theorem*
Let p be a prime and $a \varepsilon \mathbb{Z}$ such that $p \nmid a$. Then, $a^{p-1} \equiv 1 \mod p$.

Proof. Consider the following two sets of equivalence classes :

$$A = \{[a], [2a], [3a], [4a], \ldots, [(p-1)a]\}$$
$$B = \{[1], [2], [3], [4], \ldots, [p-1]\} = \mathbb{Z}_p - \{[0]\}$$

First, we want to show that $A = B$.
Clearly, $A \subseteq B$ since $p \nmid a$ and p divides none of $1, 2, 3, \ldots, p-1$.
Now, suppose that $\exists\, r, s \,\varepsilon\mathbb{N}$ such that $1 \leq r \leq s \leq p-1$ and $[ra] = [sa]$.
Then,

$$\begin{aligned} [ra] - [sa] &= 0 \\ r[a] - s[a] &= 0 \\ (r-s)[a] &= 0 \\ r - s &= 0, since [a] \neq 0 \mod p \ (since\ p \nmid a) \end{aligned}$$

But, since $r \leq p$ *and* $s \leq p$, $r - s \equiv 0 \mod p \Rightarrow r = s$.
So, $[a], [2a], [3a], \cdots, [(p-1)a]$ are all different *mod* p, which means that the cardinality of A is $p-1$.
And, since $\sharp A = p-1$, $\sharp B = p-1$ and $A \subseteq B$, we can conclude that $A = B$, that is the equivalence classes in A are congruent to the equivalence classes of B under a certain rearrangement.
Hence,

$$\begin{aligned} a * 2a * 3a * \cdots * (p-1)a &= 1 * 2 * 3 * \cdots * p - 1 \ mod\ p \\ a^{p-1} * (p-1)! &= (p-1)! \\ a^{p-1} &= 1, since (p-1)! \neq 0 \ mod\ p. \end{aligned}$$

□

Note that from a group theory point of view, considering all $a \varepsilon \mathbb{Z}$ such that $p \nmid a$ as $\mathbb{Z}_p$, this theorem is equivalent to the theorem that any element of a group elevated to the cardinality of the group must equal the identity element.

How can we use this theorem to factor an integer n ? The idea is that if $p \mid n$, and p is prime, then $a^{p-1} = 1 \mod p$, or $d = a^{p-1} - 1 = 0 \mod p$, for any a relatively prime to p, so that $gcd(d, n) = p$, the factor of n we were looking for. Obviously, we can not directly compute d because we do not know p at first. We could just compute a^m with an exponent $m = 1, 2, 3, \ldots$ until $gcd(d, n) = p$, that is until $m = p - 1$. However, that would not be more efficient than doing trial division, since we would need to execute $p - 1$ operations, each involving exponentiaton to a relatively big power.
There is however a clever way to choose m. The idea is to notice that we do not need to exponentiate a exactly to the power $m = p - 1$ since, if m is such that $p - 1 \mid m$, i.e. $m = c(p - 1)$, then

$$a^m - 1 = a^{c(p-1)} - 1 = a^{p-1^c} - 1 = 1^c \equiv 0 \mod p$$

so that $gcd(a^m, n) = p$. So, we need to choose an integer m and we will get a factor of n if $p - 1 \mid m$. By choosing m as a product of many small prime factors, the chances that this condition holds will increase.

1.2 The steps of the algorithm

1- Choose a bound B for the algorithm, usually about $10^5 - 10^6$

2- Compute

$$m = \prod_{\substack{p \ prime \\ 1 \le p \le B}} p^{\lfloor logB/logp \rfloor}$$

3- Choose a random positive integer a between 1 and n.

4- Compute $d = gcd(a, n)$.
If $d = 1$,go to 5.
If $d \neq 1$, return d. (It is a non-trivial factor of n.)

5- Compute $a^m \mod n$

6- Compute $e = gcd(a^m - 1, n)$
If $e = 1$, go to 1 and increase B.
If $e = n$, go to 3 and change a.
If $e \neq 1$ & $e \neq n$, return d. (It is a non-trivial factor of n.)

Note that the bound B defines indirectly the size of the exponent m. It hence constitutes a measure of the time the algorithm will take to factor n. A bigger B increases the probability of finding a factor of n, but it also increases the time needed to perform the algorithm.

I should also point out that some implementations of the algorithm will use $m = lcm(B)$. Then, m will still be a product of small primes, but its factorisation will not include any power of small primes, so it might fail if the factorization of $p-1$ contains a prime to a higher power.

1.3 Conditions of success of the algorithm

Pollard's algorithm will succeed to find a factor p of n if p is such that $p-1 \mid m$. Using the following definition of an x-smooth integer, we can restate this condition as follow : Pollard's $p-1$ algorithm with bound b will succeed to find a factor of n if n has a factor p such that $p-1$ is b-smooth, except in the rare cases where $p-1$ has a small prime factor to an exponent so large that it is not in the factorisation of m.

Definition 1.1. *An integer n is said to be* ***x-smooth*** *if and only if all its prime factors are smaller or equal to x.*

For example, $153 = 3^2 * 17$ is 17-smooth (and so n-smooth $\forall n \varepsilon \mathbb{N} \geq 17$).

Obviously, since n, the number to factorize, is finite, p is finite too, so there surely exists a B such that $p-1$ is b-smooth. However, if this B is too big, then Pollard's $p-1$ algorithm will not be faster than trial division. If we use Pollard's p-1 algoithm with a bound B to try to factor n, and n has a prime factor p, then the probability that we will find p is the probability that p is $B - smooth$. This probability is approximated by the probability that a number near p is $B - smooth$,which is given by $p((log(p))$ $(log(B))$ (Theorem 4.9 of [12]).

1.4 Efficiency of the algorithm

There are three possibly time-consumming steps in the Pollard's $p-1$ algorithm:

1- Compute $m = lcm(b)$

2- Compute $a^m \mod n$

3- Compute $d = gcd(a^m - 1, n)$

In this section, I will present some simple algorithms to do each of these operations efficiently.

Compute $m = lcm(b)$: Sieve of Eratosthenes

To compute m, we first need to generate a list of all the primes $leqB$. One technique to find all those primes would be to go through all the integers smaller than B and check if they are primes. Using trial divison to check primality, this technique would require $O(\sqrt(i))$ steps for each integer $2 \leq i \leq B$, for a total of $O(B\sqrt(B))$ steps. This is way too much work for only a preliminary computation.

Finding those primes can be achieved in a faster way by the Sieve of Eratosthenes. First, we write all the integers between 2 and our bound B. Then, for each number up to $\sqrt(B)$ which is not crossed out yet, we cross out all its multiples. The numbers remaining will be the integers smaller than B which have no factor smaller than $\sqrt(B)$, that is the primes smaller than B. See annex 1 for a sample code of the Sieve of Eratosthenes in Pari language.

The Sieve of Eratosthenes is very more efficient than the trivial method. The outer loop will go through all numbers smaller than $\sqrt(B)$ not yet crossed out. The exact number that is is not really important, we will just bound it above by $\sqrt(B)$. For each of these integers, we will cross out at most n p terms. So, the total needed operations will be :

$$\Sigma_{p=2}^{\sqrt(B)} \lceil \frac{B}{p} \rceil \approx \int_2^{\sqrt(B)} \frac{B}{p} dp = B \cdot ln(\sqrt(B)) = O(B \cdot log(B))$$

Moreover, it is faster to add integers and croos-out numbers than to prodeed to divisions, so the advantage of the Sieve of Erasthostenes over the trial division method is even greater than shown by this complexity analysis. As an indication of how long it takes to compute the primes, the implementation found in annex returns all the primes smaller than 500 000 in less than one second on an average computer.

Compute $a^m \mod n$: Fast modular multiplication

We will compute $a^m \mod n$ by expending m as a sum of powers of 2, repetedly squaring a, and then multiplying the relevent powers of a.

Let $m = k_0 2^0 + k_1 2^1 + \cdot + k_r 2^r$, where the $k_{i's}$ are either 0 or 1.

Then,

$$\begin{aligned} a^m &= a^{k_0 2^0 + k_1 2^2 + k_2 2^2 + \cdots + k_r 2^r} \\ a^m &= a^{k_0 2^0} * a^{k_1 2^2} * a^{k_2 2^2} * \cdots * a^{k_r 2^r} \\ a^m &= k_0 a^{2^0} * k_1 a^{2^2} * k_2 a^{2^2} * \cdots * k_r a^{2^r} \\ a^m &= A_0 * A_1 * A_2 * \cdots * A_r \end{aligned}$$

And the $A_{i's}$ are computed in the following way :

$$\begin{aligned} A_0 &= a \\ A_1 &= A_0^2 = a^2 \\ A_2 &= A_1^2 = a^4 \\ &\ldots \\ A_r &= A_{r-1}^2 = a^{2^r} \end{aligned}$$

Computing a^m as $\underbrace{a \cdot a \cdot a \cdot \cdots \cdot a}_{m \ times}$, we would need m operations, each operation consisting of one multiplication and one reduction mod n. The method just described is much more efficient. We only need r operations to computes the $A_{i's}$ and than at most r operations to add them together and get a^m. And, since $m = k_0 2^0 + k_1 2^1 + \cdot + k_r 2^r \geq 2^r$, we get that $r \leq log_2 k$. So, with this method, we can compute a^m in at most $2log_2 k$ operations, where each operation consists of one multiplication and one reduction mod n.

Compute $d = gcd(a^m - 1, n)$: Euclid's GCD algorithm

Let

$$\begin{aligned} p &= p_1^{e_1} \cdot p_2^{e_2} \cdot \cdots \cdot p_r^{e_r} \\ q &= p_1^{f_1} \cdot p_2^{f_2} \cdot \cdots \cdot p_r^{f_r} \end{aligned}$$

where $p_1, p_2, \ldots, p_r$ are distinct primes & $e_1, e_2, \ldots, e_r, f_1, f_2, \ldots, f_r \, \varepsilon \, \mathbb{N}$

Then, we can compute $gcd(p, q)$ as

$$gcd(p, q) = p_1^{min(e_1, f_1)} \cdot p_2^{min(e_2, f_2)} \cdot \cdots \cdot p_r^{min(e_r, f_r)}$$

However, this formula requires that, to compute $gcd(a^m - 1, n)$, we first find the prime factorisation of $a^m - 1$ and n. But if we had the prime factorisation of n, we would not be computing $gcd(a^m - 1, n)$ to try to factor n !

The problem of computing the gcd of two integers whose factorisation into primes we do not know has fortunately been successfully studied long ago. Indeed, Euclid published a method to find $gcd(p, q)$ as Proposition II in the second book of *The Elements* more than 2000 years ago. It is based on the following theorem :

Theorem 1.2. *GCD and remainder*
Let $a, b \, \varepsilon \, \mathbb{N}^+$ *and* $q, r \, \varepsilon \mathbb{Z}$ *such that* $a = bq + r$. *Then,* $gcd(a, b) = gcd(r, b)$.

Proof. Let $d = gcd(a, b)$ and $e = gcd(r, b)$.
We will show that $d \leq e$ and $e \leq d$, so that $d = e$.

$$d \mid a \ \& \ d \mid b \Rightarrow \ d \mid a - bq = r$$

So, d is a common divisor of b and r. And, since $e = gcd(b, r)$, $d \leq e$.
Also,

$$e \mid r \ \& \ e \mid b \Rightarrow \ e \mid bq - r = a$$

So, e is a common divisor of a and b. And, since $d = gcd(a, b)$, $e \leq d$.
Hence, $d = e$, i.e $gcd(a, b) = gcd(b, r)$. □

Euclid's algorithm consists of reducing the problem of computing $gcd(a, b)$ of size a to the problem of computing $gcd(b, r = a - qb)$ of size b by the division algorithm and then applying the same technique until we get the $gcd(a, b)$:

$$\begin{aligned} a &= bq_1 + r_1 \quad 0 \leq r_1 < b \\ b &= r_1 q_2 + r_2 \quad 0 \leq r_2 < r_1 \\ &\ldots \\ r_{n-1} &= r_n q_{n+1} + r_{n+1} \quad 0 \leq r_{n+1} < r_n \\ r_n &= r_{n+1} q_{n+2} \end{aligned}$$

Since the sequence of remainders is strictly decreasing and the remainders are all strictly positive, there will necessarily eventually be one remainder equal to 0. In our example, it was r_{n+2}, and so

$$gcd(a, b) = gcd(b, r_1) = \ldots = gcd(r_n, r_{n+1}) = r_{n+1}$$

Euclid's algorithm is really simple to implement recursively. Here is my code of the Euclidean algorithm in Pari :

```
Input  : two integers a and b
Output : gcd(a,b)
Note   : In pari, the notation is if(condition, then, else)

Gcd(a,b) =
{
  if(b==0, return(a), return(Gcd(b, a%b)));
}
```

Let's now analyse how many steps will be required to compute $gcd(a, b)$ with Euclid's algorithm.

Lemma 1.1. *Every two steps of Euclid's algorithm, the remainder is at least halved i.e.* $r_{n+1} \leq \frac{1}{2} r_{n-1}$

Proof. We know that $r_{n-1} = r_n q_{n+1} + r_{n+1}$, so that $r_{n+1} = r_{n-1} - r_n q_{n+1}$.
Also, $r_{n+1} < r_n$. So, if $r_n \leq \frac{1}{2} r_{n-1}$, we are done.
Hence, we can suppose that $r_n > \frac{1}{2} r_{n-1}$
Then,

$$r_{n+1} = r_{n-1} - r_n q_{n+1} < r_{n-1} - \frac{1}{2} r_{n-1} q_{n+1} = r_{n-1}(1 - \frac{1}{2} q_{n+1})$$

.
Now, $q_{n+1} \neq 0$, since otherwise we would have $r_{n+1} = r_{n-1}$, $but r_{n+1} < r_n < r_{n-1}$. Since q_{n+1} is a non-zero integer, $q_{n+1} \geq 1$, so that

$$r_{n+1} < r_{n-1}(1 - \frac{1}{2} q_{n+1}) < \frac{1}{2} r_{n-1}$$

□

Theorem 1.3. *Euclidean algorithm complexity*
The Euclidean algorithm returns $gcd(a, b)$ *in at most* $2log_2(2b)$*steps.*

Proof. From lemma 1.1, we have $r_{n+1} \leq \frac{1}{2} r_{n-1}$. Since $r_1 < b$,

$$r_3 < \frac{1}{2} b, \quad r_5 < \frac{1}{2} r_3 < \frac{1}{4} b, \quad \cdots, r_{2n-1} < \frac{1}{2^{n-1}}$$

So, the algorithm terminates as soon as $2^{n-1} \geq b$, since then $r_{2n-1} \leq 1$, so $r_{2n-1} = 0$ since it's a non-negative integer.
Hence, when the gcd is found, we have

$$n - 1 \geq log_2 b \quad i.e. \quad n \geq 1 + log_2 b = log_2(2b)$$

So, the Euclidean algorithm takes at most $2log_2(2b)$ to find $gcd(a, b)$. □

1.5 Possible improvements of the algorithm

When the bound B becomes so big that it is too expensive to continue doing Pollard's algorithm as describe earlier, there is still a possibility to go on and factor n, which is a second-stage to the algorithm. At this point, we change the notation of B to $B1$, and introduce a new bound $B2$, usually $100B1$ or $10000B1$. (This actually depends of our implementation, but to increase our chances of finding a factor, it is suggested in [7] to choose $B2$ such that our algorithm will spend as much time in the second phase as it spend in the first one.)

So, in the second stage, we keep the value a^Q, $where\ Q = lcm(primes \leq B1)$ and we then compute $a^{Qq_1}, a^{Qq_2}, \cdots, a^{Qq_r}$, $where\ q_1, q_2, \cdots, q_r$ are all the primes between $B1$ and $B2$, and check each time $gcd(a^Q q_i - 1, n)$ just as in the last step of the first stage. To compute a^{Qq_s} from a^{Qq_r}, compute $a^{Q(q_s - q_r)}$. Since the difference between two primes is usually not too big, this is done efficiently by precomputing $a^{2Q}, a^{4Q}, \cdots, a^{Q2n}$ with n being a few hundreds before starting stage 2.

The second stage of Pollard's algorithm will find a factor p if its $p-1$ prime factors are all $\leq B1$ except for one which is between $B1$ and $B2$. Notice that it will not find a factor p if $p-1$ as more than one factor between $B1$ and $B2$ because the prime factors are included only one at a time in the exponent, except in a few particuliar cases.

Some other technical improvement can also make the algorithm run faster. For example, we could compute the gcd's only once a few hundred times, and then go backwards if more than one factor has been found. Some details on other speeding improvements can be found in [5]. However, even with all these improvements, Pollard's algorithm will not factor n if $p-1$ only has big primes in its factorization. At this point, it is better to switch to an other algorithm, for example Lenstra's factoring method which extends the idea of the $p-1$ algorithm to the group of points on an elliptic curve.

2 Lenstra's algorithm using elliptic curves

Before we study Lenstra's algorithm for factoring integers with elliptic curves, we need to take a look at a few properties of elliptic curves.

2.1 Elliptic curves

Definition

An elliptic curve is defined by a polynomial of degree threee of two variables, $f(x, y) = 0$, where x and y are taken from an arbitrary field. By an appropriate change of variables, any elliptic curve can be reduced to the following Weierstrass form : $y^2 = x^3 + ax + b$, provided the field over which the elliptic curve is defined does not have characteristic 2 or 3. We have to exclude these cases because transforming a general elliptic curve into the Weierstrass form involves division by 2 and by 3, which then would not be allowed. Since the algorithm does not require any elliptic curve over a field of characteristic 2 or 3, we will not consider them, and concentratre only on elliptic curves in the Weierstrass form, as this simplifies considerably our work. So, our definition of an elliptic curve is as follow :

Definition 2.1. *An elliptic curve is defined by a non-singular equation of the form* $y^2 = x^3 + ax + b$ *where* $a, b, x, y \varepsilon K$, *and* K *is a field with characteristic* $\neq 2, 3$.

Non-singularity of the curve only means that it does not have any repeated roots, that is its graph does not have any cusps or self-intersections. Just as the discriminant $b^2 - 4ac$ of a quadratic equation $ax^2 + bx + c$ vanishes if the equation has a repeated root, the discriminant $\Delta = -16(4a^3 + 27b^2)$ of an elliptic curve will vanish if the curve is non-singular. So, we check for non-singularity by adding the condition that $4a^3 + 27b \neq 0$.

One interesting geometric property of a curve as defined in 2.1 is that any line crossing the curve at two points on the plane will necessarily meet it at a third point. To see that, suppose that the line is given by $y = mx + b$ and look at its intersections with the curve $y^2 = x^3 + ax + b$. The points which are both on the line and the curve will satisfy the equation

$$y^2 = (mx + b)^2 = mx^2 + 2bm + b^2 \;=\; x^3 + ax + b$$

And,this is a cubic equation in x, so it has either 3 real roots or only 1.

Group law on elliptic curves

We are now interested in defining an operation on the set of points on an elliptic curve that will transform it into a group. The operation on the group of points on the elliptic curve should take as input two points on the curve, and then return a third point on the curve. An obvious way to do that would be to use the property just mentionned and draw a line between the two known points and then take the third inteserction point of this line with the curve. Let's denote the result of this operation with the points P and Q by $P * Q$. Clearly the * operation does not transform the set of points on the elliptic curve into a group ; there is for example no identity element. However, we will use this * to define the group operation. But first, let's define the set of points on the elliptic curve :

Definition 2.2. *Let E be an elliptic curve defined over a field K.*
Then, the set of points $E(K)$ on the elliptic curve is defined as

$$E(K) = \{(x, y) s.t. x, y \varepsilon K \text{ and } y^2 = x^3 + ax + b\} \ U \ O_E$$

where O_E denotes a point at infinity.

Note that the point O_E does not appear on the graph of $E(K)$, but has to be imagined to be at infinity. It is the third point that any vertical line on the curve will meet. We include it in our set of points on the elliptic curve because we will use it as our identity element.

Definition 2.3. *Let P and Q be points on the elliptic curve E.*
Then, $P + Q = O_E * (P * Q)$.

Theorem 2.1. *Group operation on the curve*
The addition defined in 2.2 transforms $E(K)$ into a group.

Proof. 1- O_E is the identity element
This states that $P + O_E = O_E * (P * O_E) = P$
We can see that graphically. $P * O_E$ gives us the third point on the curve and the line between P and O_E, which is just a vertical line going trough P. Then, when we star this point with O_E, we get the third point on the line between O_E and $P * O_E$, that is P.

2- Let $P \ \varepsilon \ E(K)$. Then, $\exists$ a $P' \ \varepsilon \ E(K)$ st. $P + P' = P' + P = O_E$. Define P' as the reflexion about the x-axis of the point P. Then P' is on the curve, since $(x, y) \ \varepsilon \ E(K) \Rightarrow (x, -y) \ \varepsilon \ E(K)$. Now, $P + P'$ is the third point on a vertical line going through P and P', that is O_E.

3- Let $P, Q, R \, \varepsilon \, E(K)$. Then, $(P+Q)+R = P+(Q+R)$. Associativity is not as easy to prove graphically (although [10] gives an idea of how to do so), but it can be proven easily using the algebric formulas for adding points developped in the next section. An interested reader can work out this long and tedious proof by himself or look at [9] for a more conceptual proof.

□

Formulas of the group addition

In order to work efficiently with points on elliptic curves, we have to be able to add them together without relying on a graph of an elliptic curve an drawing lines. We need to describe algebraicly the operation of addition on the group of points on an elliptic curve.

Let $P_1 = (x_1, y_1),\ P_2 = (x_2, y_2),\ P_1 * P_2 = (x_3, y_3)$.
Then, $P1 + P2 = (x_3, -y_3)$ since the third intersection point with the curve of a line passing through a point P and the point at infinity is the reflexion of the point P on the $x - axis$.
So, now we need to find the value of x_3 and y_3. The line between P_1 and P_2 is given by the equation

$$y = mx + c,\ where\ m = \frac{y_2 - y_1}{x_2 - x_2}\ and\ c = y_1 - mx_1 = y_2 - mx_2$$

So, when this line intersects the cubic, we have

$$y^2 = (mx + c)^2 = x^3 + ax + b$$

Which can be rewritten as

$$0 = x^3 - m^2x^2 + (a - 2bx)x + (b - c^2)$$

We know that this equation has the three roots $x_1, x_2\ and\ x_3$, so that

$$\begin{aligned} x^3 - m^2x^2 + (a - 2bx)x + (b - c^2) &= (x - x_1)(x - x_2)(x - x_3) \\ &= x^3 - (x_1 + x_2 + x_3)x^2 \\ &\quad +(x_1x_2 + x_1x_3 + x_2x_3)x - x_1x_2x_3 \end{aligned}$$

Hence,

$$m^2 = x_1 + x_2 + x_3,\ so\ x_3 = m^2 - x_1 - x_2$$

And, $y_3 = mx_3 + c$.

But, in the case where $P_1 = P_2$, we can not use these equations since computing m will involve a division by zero. In this case, we find the slope m by computing the tangent of the curve at the point $P_1 = P_2$:

$$y^2 = f(x) = x^3 + ax + b$$

$$So,\ m = \frac{dy}{dx} = \frac{f'(x)}{2y} = \frac{3x^2 + a}{2y}$$

Hence, we can add two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ on the elliptic curve $y^2 = x^3 + ax + b$ in the following way :

$$P_1 + P_2 = \begin{cases} \mathbb{O}_{\mathbb{E}} & \text{if } P_1 = -P_2 \\ (m^2 - x_1 - x_2, m(x_1 - x_3) - y_1) & \text{otherwise} \end{cases}$$

where

$$m = \begin{cases} \frac{3x_1^2 + a}{2y_1} & \text{if } P_1 = P_2 \\ \frac{y_2 - y_1}{x_2 - x_1} & \text{otherwise} \end{cases}$$

2.2 General idea of the algorithm

Recall that in Pollard's p-1 algorithm, we can not factor n if it doesn't have a prime factor p with $p - 1$ smooth. H.W. Lenstra developped in 1985 [4] a new factorisation algorithm which avoid this problem b use the group of random elliptic curve over the field $\mathbb{Z}_p$ instead than the multiplicative group of Z_p. The idea is that the later always has order $p - 1$, but the order of the group on an elliptic curve varies with the curve, so that if the order is not smooth enough, we can just change curve.

Since we do not know p at first, we can not define an elliptic curve over the field $\mathbb{Z}_p$. So, we define the elliptic curve over the ring $\mathbb{Z}_n$. Technically an elliptic curve modulo n is not really an elliptic curve. But, by the Chinese remainder theorem, what goes on $modn$ can be imagined seperatly $modp$ and $modq$ for factors p and q of n. By adding the condition that for the elliptic curve $gcd(4a^3 + 27b^2, n) = 1$, we ensure that it is actually an elliptic curve mod p and $modq$.

Now, we want to find a number k which is congruent to 0 mod p or mod q, but only for one at a time, so that $gcd(k, n)$ is a non-trivial factor of n. To find this number,we add a point with himself a certain number of times using the formulas derived earlier. We can do that even if we do not

work over a ring because most residues $\mod n$ have inverses, so that we can correctly compute the sum of two points on this curve. The addition formulas will only break if we find a non-invertible element in $\mathbb{Z}_n$, but then we will have factored n since non-invertible elements correspond to elements whose gcd with n is not 1.

2.3 The steps of the algorithm

Here are the steps of the algorithm, given an integer n to factor :

1- Choose a bound B for the algorithm, usually about $10^5 - 10^6$

2- Compute

$$k = \prod_{\substack{p \ prime \\ 1 \leq p \leq B}} p^{\lfloor logB/logp \rfloor}$$

3- Choose random integers $a, x, y \varepsilon \mathbb{Z}_n$. Then, $b = y^2 - x^3 - ax$. Start over the step 3 until $gcd(4a^3 + 27b^2, n) \neq 1$, $E : y^2 = x^3 + ax + b$ is an elliptic curve. Then, you have a random elliptic curve E and a point $P = (x, y)$ on it.

4- Compute $kP = \underbrace{P + P + \cdots + P}_{k \ times}$.
For each computation, we will either get a new point on the curve, or a factor of n.

5- If a factor of n was not found in 4, go back to 3 and make a new choice for for the point and
or the curve, or go to one and increase B.

Note that in step 3, we choose x, y *and* a and then compute b instead of choosing first an elliptic curve and then finding a point on it because that would require taking the square root of an element of $\mathbb{Z}_n$, a problem equivalent to that of factoring n.

2.4 Conditions of success of the algorithm

The success of the algorithm now depends of the order of the random elliptic curve chosen. If the order of an elliptic curve divides k, than kP will this identity element. Since k is again a product of small primes, the probability that this holds is the probability that given a bound B, a random elliptic curve is $B - smooth$. First, we have to know approximately what the order of the elliptic curve will be $mod\ p$:

Theorem 2.2. *Hasse's Theorem*
Let E be an elliptic curve mod p. Then,

$$p+1-2\sqrt(p) \ < \ \#E \ < p+1+2\sqrt(p)$$

Proof. The proof of this theorem is not in the scope of this text. You can find it in [9]. Note that it was also shown by Deuring [2] that every integer in this interval will actually be the order of an elliptic curve mod p, and by Lenstra [4] that if the curves are chosen at random, then their orders are well distributed in this interval. □

Unfortunately, we can not conclude that any $p+1-2\sqrt(p) \ < \ \#E(\mathbb{Z}_p) \ < \ p+1+2\sqrt(p)$ is $B-smooth$. But, if we make the assumption that N can be chosen from a longer interval (say $p/2 < N < 3p/2$), then, according to [12], it would follow from a theorem of Canfield et al. [1] that the probability that N is smooth is u^{-u} where $u = \frac{lnp}{lnB}$.

2.5 Efficiency of the algorithm

Implementation optimisation

The implementing difficulties of this algorithm are similar to the ones of Pollard's algorithm, and so are the solutions. To get the list of all primes smaller or equal to B, we will use the the Sieve of Eratosthenes, as presented in section 1.4.1. Also, the fast exponentiation algorithm, with multiplication replaced by addition of points on the elliptic curve, will allow us to compute efficiently kp.
The only difference is that is Lenstra's algorithm requires that we proceed to divisions mod n for the addition operation. Hence, we need to find inverses of elements in $\mathbb{Z}_n$. But not every element in $\mathbb{Z}_n$ is invertible. So, we need an algorithm that will tell us if the element is invertible or not and, in the latter case, give us its inverse. We could use the Euclidean algorithm since it would give us $gcd(elt, n)$ and, be going through the equations of the algaorithm backwards we could find r, *and* s such that $r*elt+s*n = gcd(elt, n)$. So, if $gcd(elt, n)$ is one, then r is the inverse of elt. An algorithm which do exactly that is called the Extended Euclidean Algorithm. Here is this algorithm in Pari language :

```
Input : a,b two integers
Output : a vector [g,x,y]
         where g=gcd(a,b) and x & y are s.t ax+by=gcd(a,b)
```

```
Egcd(a,b) =
{
  local(x,y,g,r,s,t,u,v,w,q,ans);
  x=1; y=0; g=a;
  r=0; s=1; t=b;
  ans=vector(3);
  while(t > 0,
    q=floor(g\t);
    u=x-q*r; v=y-q*s; w=g-q*t;
    x=r;     y=s;     g=t;
    r=u;     s=v;     t=w;
    );
  ans[1]=g; ans[2]=x; ans[3]=y;
  return(ans);
}
```

We can see why this algorithm works by concentrating on what happens to the variables $g, t,$ *and* w. During the algorithm, w gets the value of g mod t, g becomes t and w becomes g. These changes correspond to the ones of the variables $a, b,$ *and* r in the simple Euclidean algorithm. Thus, since g and t are initialized to the values a and b, g is equal to $gcd(a, b)$ at the end of the algorithm. We can also see that x and y are the correct values since during the entire algorithm both equations $ax + by = g$ and $ar + bs = t$ stay true. This is because the assignments in the second line correspond to substracting q times the second equation from the first.

Complexity analysis

To analyse the complexity of the elliptic curve algorithm, we need a few more theoretical results. First, here is a theorem which gives some indication on the distribution of prime numbers :

Theorem 2.3. *Prime Number Theorem Let x be a positive integer. Let $\pi(x)$ be the number of prime numbers less than or equal to x.*
Then, $\lim_{n\to\infty} \frac{\pi(x)}{\frac{x}{lnx}} = 1$.

Proof. This theorem was conjectured more than 200 years ago by Gauss, but wasn't proved until 1896. There exits different proofs of the Prime Number Theorem, but they either require some high-level mathematics or are too complicated to be included here. □

Now, we are ready to prove the following theorem from [12] which gives an expected value for the number of group operations needed to factor n with the elliptic curve method.

Theorem 2.4. *Complexity of the elliptic curve method*
Let $n\varepsilon\mathbb{Z}^+$, p be a prime factor of n and B be the optimal bound for finding p with the elliptic curve method.
Then, $B = L(p)^{\frac{\sqrt{(2)}}{2}}$, the expected number of group operations to find p is $L(p)^{\sqrt{(2)}}$, and the expected number of group operations to find one prime factor of n is $L(n)$, where $L(x) = e^{\sqrt{(lnxlnlnx)}}$.

Proof. The bound B is optimal if it minimizes the number of group operations to find p. By the Prime Number Theorem, we know that there is approximately $\frac{B}{lnB}$ primes $\leq B$. Now, since for almost all of the primes q less than or equal to B, $q^e \leq B$ has $e = 1$, the algorithm used to compute $(q^e)P$ will take about $logB$ group additions. Hence, the total of group additions done per curve is about $(B/lnB)lnB = B$, given that we work with bound B.
From section 2.4, we know that we will need to try $\frac{1}{u^{-u}} = u^u$ curves to factor n, where $u = \frac{lnp}{lnB}$. So, with bound B, the expected total number of operations to factor n is $f(B) = Bu^u$. Now, we need to find the value of B that will minimize $f(B)$.

Let $a = \frac{lnb}{lnL(p)}$ so that $B = (L(p))^a$. Then,

$$lnB = aln(L(p)) = a\sqrt{lnplnlnp}$$

So,

$$u = \frac{lnp}{lnB} = \frac{lnp}{a\sqrt{lnplnlnp}} = \frac{1}{a}\sqrt{\frac{lnp}{lnlnp}}$$

and

$$lnu = \frac{1}{2}lnlnp - \frac{1}{2}lnlnlnp - lna \approx \frac{1}{2}lnlnp$$

Hence,

$$ulnu = \frac{1}{a}\sqrt{\frac{lnp}{lnlnp}}\frac{1}{2}lnlnp = \frac{1}{2a}lnL(p)$$

This leads to

$$u^u = e^{ulnu} \approx L(p)^{\frac{1}{2a}}$$

Now, the function $f(B)$ that we wanted to minimize can be expressed as

$$f(B) = Bu^u \approx L(p)^a L(p)^{\frac{1}{2a}} = L(p)^{1+\frac{1}{2a}}$$

Now, since $L(p)$ is a positive constant for $p \geq e^e$, the minimum of $f(B)$ occurs when $a + \frac{1}{2a}$ is minimal. From calculus, we find that this happens when $a = \frac{\sqrt{2}}{2}$ and that the value of $a + \frac{1}{2a}$ is then $\sqrt{(2)}$.
Hence, the optimal value of B for this algorithm is $L(p)^a = L(p)^{\frac{\sqrt{(2)}}{2}}$.
Then, the expected total group operations to factor n is $f(B) = L(p)^{\sqrt{(2)}}$.
But, since we do not know p when we start the algorithm, we would rather have an estimate for the number of group operations needed with respect to n. Well, considering p as the smallest factor of n, we know that $p \leq \sqrt{(n)}$ so that $lnp \leq \frac{1}{2} lnn$ and $lnlnp < lnlnn$, so the expected total number of group additions to factor n is

$$L(p)^{\sqrt{(2)}} = e^{\sqrt{(2lnplnlnp)}} < e^{\sqrt{(lnnlnlnn)}} = L(n)$$

□

Note that in our complexity analysis, we assumed that the optimal value of B was used, but this optimal value can not be computed at first because it depends on p, which is not known. However, if we slowly increase B when using the algorithm, then it will act as if we were using the optimal value of B. For some details on how exactly to increase the value of B see [11].

2.6 Possible improvements of the algorithm

Lenstra's algrithm also admits a second stage just like Pollard's one that will factor n if it has a prime factor p such that $p - 1$ has all its prime factors smaller than $B1$ except possibly one between $B1$ and $B2$. Again, some other technical improvement can also make the algorithm run faster. For example, using projective coordinates diminishes the work needed to compute the sums of points on $E(K)$. Some details on other speeding improvements can be found in [5].

APPENDIX I - Sieve of Eratosthenes

```
Input : B, a positive integer
Output : pri, a list of all primes smaller or equal than B
Note : The following functions already implemented in Pari are used :
       vector(n)    creates an empty vector of length n
       sqrt(n)      returns the square root of n
       length(a)    returns the length of the vector a

PrimesEratos(B) =
{
  local(int,p,i,j,pri,count);
  int = vector(B);
  int[1] = 1; \\since 1 isn't prime
  p = 2; i = 1;
while( p <= sqrt(B),
  i = 2*p;
  while( i <= length(int),
  int[i] = 1;
  i = i + p;
    );
    p = p + 1;
    while(int[p] == 1, p = p + 1);
  );
  i = 1; j = 1;
  count = 0;
  while(i < = length(int),
    if(int[i] == 0, count = count + 1);
    i = i + 1;
    );
  pri = vector(count);
  i = 1;
  while(i < = length(int),
    if(int[i] == 0, pri[j] = i ; j = j + 1);
    i = i + 1;
    );
  return(pri);
}
```

References

[1] E. Canfield, P. Erdos, C. Pomerance, *On a problem of Oppenheim concerning "factorisatio numerorum."* J. Number Theory, 17 (1983), 1-28.

[2] M. Deuring, *Die Typen der Multiplikatorenringe elliptischer Funktionenkorper* Abh. Math. Sem. Hansischen Univ., 14 (1941), 197-272.

[3] D. Harel, *Algorithmics - The Spirit of Computing* 3rd edition, Addison Wesley : Pearson Education, New-York, 2004.

[4] H. W. Lenstra Jr, *Factoring Integers with Elliptic Curves* Annals of Mathematics, 126 (1987), 649-673.

[5] P. L. Montgomery, *Speeding the Pollard and elliptc curve methods of factorization*, Math. Comp. 48 (1987), 243-264.

[6] J. M. Pollard, *Theorems on factorization and primality testing* Proc. Cambridge Philos. Soc., 76 (1974), 521-528.

[7] H. Risel, *Prime Numbers and Computer Method Factorization* Birkhauser, Boston, Massachusetts, Second editon, 1994.

[8] R. L. Rivest, A. Shamir and L. Adleman, *A Method for Obtaining Digital Signatures and Public Key Cryptosystems* Communications of the ACM, 21, 2(1978), 120-126.

[9] J. H. Silverman, *The Arithmetic of Elliptic Curves*, Graduate Texts in Mathematics 106, Springer-Verlag, 1994.

[10] J. H. Silverman and J. Tate, *Rational Points on Elliptic Curves*, Graduate Texts in Math. 106, Springer-Verlag, New-York 1986.

[11] R. D. Silverman and S.S Wagstaff Jr *A practical analysis of the elliptic curve factoring algorithm* Math. Comp., 61 (1993), 445-462.

[12] S. S. Wagstaff Jr, *Cyptanalysis of Number Theoretic Ciphers* Computational Mathematics Series, Chapman & Hall/CRC, Boca Raton, 2003.

[13] Y.Y Song, *Number Theory for Computing* 2nd edition, Springer-Verlag, Berlin, New-York, 2002.