18.783 Elliptic Curves **Spring 2015**
Lecture #9 **03/05/2015**

9 Schoof's algorithm

In the early 1980's, René Schoof [3, 4] introduced the first polynomial-time algorithm to compute $\#E(\mathbb{F}_q)$. Extensions of Schoof's algorithm remain the point-counting method of choice when the characteristic of $\mathbb{F}_q$ is large (e.g., when q is a cryptographic size prime).[1]

Schoof's basic strategy is very simple: compute the the trace of Frobenius t modulo many small primes ℓ and use the Chinese remainder theorem to obtain t; this then determines $\#E(\mathbb{F}_q) = q + 1 - t$. Here is a high-level version of the algorithm.

Algorithm 9.1. Given an elliptic curve E over a finite field $\mathbb{F}_q$ compute $\#E(\mathbb{F}_q)$ as follows:

1. Initialize $M \leftarrow 1$ and $t \leftarrow 0$.
2. While $M \leq 4\sqrt{q}$, for increasing primes $\ell = 2, 3, 5, \ldots$ that do not divide q:
   a. Compute $t_\ell = \operatorname{tr}\pi \bmod \ell$.
   b. Update $t \leftarrow \big((M^{-1} \bmod \ell)Mt_\ell + (\ell^{-1} \bmod M)\ell t\big) \bmod \ell M$ and $M \leftarrow \ell M$.
3. If $t > M/2$ then set $t \leftarrow t - M$.
4. Output $q + 1 - t$.

Step 2b uses an iterative version of the Chinese remainder theorem to ensure that

$$t \equiv \operatorname{tr}\pi_E \bmod M$$

holds throughout.[2] Once M exceeds $4\sqrt{q}$ (the width of the Hasse interval), t uniquely determines the integer $\operatorname{tr}\pi_E$, whose absolute value must be less than $M/2$; this allows the sign of t to be determined in step 3. The key to the algorithm is the implementation of step 2a, which is described in the next section, but let us first consider the primes ℓ that the algorithm uses.

Let ℓ_{max} be the largest prime ℓ for which the algorithm computes t_ℓ. The Prime Number Theorem implies[3]

$$\sum_{\text{primes } \ell \leq x} \log \ell \sim x,$$

so $\ell_{max} \approx \log 4\sqrt{q} \approx \frac{1}{2}n = O(n)$, and we need $O\big(\frac{n}{\log n}\big)$ primes ℓ (as usual, $n = \log q$). The cost of updating t and M is bounded by $O(\mathsf{M}(n)\log n)$, thus if we can compute t_ℓ in time polynomial in n and ℓ, then the whole algorithm will run in polynomial time.

[1] There are deterministic p-adic algorithms for computing $\#E(\mathbb{F}_q)$ that are faster than Schoof's algorithm when the characteristic p of $\mathbb{F}_q$ is very small; see [2]. But their running times are exponential in $\log p$.

[2] There are faster ways to apply the Chinese remainder theorem; see [1, §10.3]. They are not needed here because the complexity is overwhelmingly dominated by step 2a.

[3] In fact we only need Chebyshev's Theorem to get this.

9.1 Computing the trace of Frobenius modulo 2.

We first consider the case $\ell = 2$. Assuming q is odd (which we do), $t = q + 1 - \#E(\mathbb{F}_q)$ is divisible by 2 if and only if $\#E(\mathbb{F}_q)$ is divisible by 2, equivalently, if and only if $E(\mathbb{F}_q)$ contains a point of order 2. If E has Weierstrass equation $y^2 = f(x)$, the points of order 2 in $E(\mathbb{F}_q)$ are those of the form $(x_0, 0)$, where $x_0 \in \mathbb{F}_q$ is a root $f(x)$. So $t \equiv 0 \bmod 2$ if $f(x)$ has a root in $\mathbb{F}_q$, and $t \equiv 1 \bmod 2$ otherwise.

One minor issue worth noting is that the algorithm we saw in Lecture 4 for finding roots of polynomials over finite fields is probabilistic, and we would like to determine $t \bmod 2$ deterministically. But we don't actually need to *find* the roots of $f(x)$ in $\mathbb{F}_q$, we just need to determine whether any exist. For this we only need the first step of the root-finding algorithm, which computes $g = \gcd(x^q - x, f(x))$. The degree of g is the number of distinct roots of f, so $t \equiv 0 \bmod 2$ if and only if $\deg g > 0$. This is a deterministic computation, and since the degree of f is fixed, it takes just $O(n\mathsf{M}(n))$ time.

This addresses the case $\ell = 2$, so we now assume that ℓ is odd.

9.2 The characteristic equation of Frobenius modulo ℓ

Recall that the Frobenius endomorphism $\pi\,(x, y) = (x^q, y^q)$ has the characteristic equation

$$\pi^2 - t\pi + q = 0$$

in the endomorphism ring $\operatorname{End}(E)$, where $t = \operatorname{tr}\pi$ and $q = \deg\pi$. If we restrict π to the ℓ-torsion subgroup $E\,[\ell]$, then the equation

$$\pi_\ell^2 - t_\ell\pi_\ell + q_\ell = 0 \tag{1}$$

holds in $\operatorname{End}(E[\ell])$, where $t_\ell \equiv t \bmod \ell$ and $q_\ell \equiv q \bmod \ell$ may be viewed either as restrictions of the scalar multiplication maps $[t]$ and $[q]$, or simply as scalars in $\mathbb{Z}/\ell\mathbb{Z}$ multiplied by the restriction of the identity endomorphism $[1]_\ell$. We shall take the latter view, regarding q_ℓ as $q_\ell[1]_\ell$, the sum of q_ℓ copies of $[1]_\ell$, which we can compute using double-and-add, once we know how to explicitly add elements of $\operatorname{End}(E[\ell])$.

A key fact we need is that equation (1) not only uniquely determines t_ℓ, it suffices to verify (1) at any nonzero point $P \in E[\ell]$.

Lemma 9.2. *Let $E/\mathbb{F}_q$ be an elliptic curve with Frobenius endomorphism π, let ℓ be a prime, and let P be a non-trivial point in $E[\ell]$. Suppose that for some integer c the equation*

$$\pi_\ell^2(P) - c\pi_\ell(P) + q_\ell(P) = 0$$

holds. Then $c \equiv \operatorname{tr}\pi \bmod \ell$.

Proof. Let $t = \operatorname{tr}\pi$. From equation (1) we have

$$\pi_\ell^2(P) - t\pi_\ell(P) + q_\ell P = 0,$$

and we are given that

$$\pi_\ell^2(P) - c\pi_\ell(P) + q_\ell P = 0.$$

Subtracting these equations, we obtain $(c - t)\pi_\ell(P) = 0$. Since $\pi_\ell P$ is a nonzero element of $E[\ell]$ and ℓ is prime, the point $\pi_\ell(P)$ has order ℓ, which must divide $c - t$. So $c \equiv t \bmod \ell$. □

Let $h = \psi_\ell$ be the ℓth division polynomial of E; since ℓ is odd, we know that ψ_ℓ does not depend on the y-coordinate, so $h \in \mathbb{F}_q[x]$. As we proved in Lecture 6, a nonzero point $P = (x_0, y_0) \in E(\overline{\mathbb{F}}_q)$ lies in $E[\ell]$ if and only if $h(x_0) = 0$. Thus when writing elements of $\operatorname{End}(E[\ell])$ as rational maps, we can treat the polynomials appearing in these maps as elements of the ring

$$\mathbb{F}_q\left[x, y\right]/(h(x), y^2 - f(x)),$$

where $y^2 = f(x) = x^3 + Ax + B$ is the Weierstrass equation for E.

In the case of the Frobenius endomorphism, we have

$$\begin{aligned}\pi_\ell &= \big(x^q \bmod h(x),\ y^q \bmod (h(x), y^2 - f(x))\big)\\ &= \Big(x^q \bmod h(x), \big(f(x)^{(q-1)/2} \bmod h(x)\big)y\Big), \end{aligned}\tag{2}$$

and similarly,

$$\pi_\ell^2 = \Big(x^{q^2} \bmod h(x), \big(f(x)^{(q^2-1)/2} \bmod h(x)\big)y\Big).$$

We also note that

$$[1]_\ell = (x \bmod h(x), \big(1 \bmod h(x)\big)y).$$

Thus we can represent all of the nonzero endomorphisms that appear in equation (1) in the form $(a(x), b(x)y)$, where a and b are elements of the polynomial ring $R = \mathbb{F}_q[x]/(h(x))$.

Let us consider how to add and multiply elements of $\operatorname{End}(E[\ell])$ that are represented in this form.

9.3 Multiplication in End($E[\ell]$).

Recall that we multiply endomorphisms by composing them. If $\alpha_1 = \big(a_1(x), b_1(x)y\big)$ and $\alpha_2 = \big(a_2(x), b_2(x)y\big)$ are two elements of $\operatorname{End}(E[\ell])$, then the product $\alpha_1\alpha_2$ in $\operatorname{End}(E[\ell])$ is simply the composition

$$\alpha_1 \circ \alpha_2 = (a_1(a_2(x)),\ b_1(a_2(x))b_2(x)y),$$

where we may reduce $a_3(x) = a_1(a_2(x))$ and $b_3(x) = b_1(a_2(x))b_2(x)$ modulo $h(x)$.

9.4 Addition in End($E[\ell]$).

Recall that addition of endomorphisms is defined in terms of addition on the elliptic curve. Given $\alpha_1 = \big(a_1(x), b_1(x)y\big)$ and $\alpha_2 = \big(a_2(x), b_2(x)y\big)$, to compute $\alpha_3 = \alpha_1 + \alpha_2$, we simply apply the formulas for point addition. Recall that the general formula for computing a nonzero sum $(x_3, y_3) = (x_1, y_1) + (x_2, y_2)$ on the elliptic curve $E\colon y^2 = x^3 + Ax + B$ is

$$x_3 = m^2 - x_1 - x_2, \qquad y_3 = m(x_1 - x_3) - y_1,$$

where

$$m = \begin{cases} \frac{y_1 - y_2}{x_1 - x_2} & \text{if } x_1 \neq x_2 \\ \frac{3x_1^2 + A}{2y_1} & \text{if } x_1 = x_2. \end{cases}$$

Applying these to $x_1 = a_1(x), x_2 = a_2(x), y_1 = b_1(x)y, y_2 = b_2(x)y$, when $x_1 \neq x_2$ we have

$$m(x, y) = \frac{b_1(x) - b_2(x)}{a_1(x) - a_2(x)}y = r(x)y,$$

where $r = (b_1 - b_2)/(a_1 - a_2)$, and when $x_1 = x_2$ we have

$$m(x,y) = \frac{3a_1(x)^2 + A}{2b_1(x)y} = \frac{3a_1(x)^2 + A}{2b_1(x)f(x)}y = r(x)y,$$

where now $r = (3a_1^2 + A)/(2b_1 f)$. Noting that $m(x,y)^2 = (r(x)y)^2 = r(x)^2 f(x)$, the sum $\alpha_1 + \alpha_2 = \alpha_3 = (a_3(x),\ b_3(x)y)$ is given by

$$\begin{aligned} a_3 &= r^2 f - a_1 - a_2,\\ b_3 &= r(a_1 - a_3) - b_1. \end{aligned}$$

In both cases, provided that the denominator of r is invertible in the ring $R = \mathbb{F}_q[x]/(h(x))$, we can reduce r to a polynomial modulo h and obtain $\alpha_3 = \big(a_3(x), b_3(x)y\big)$ in our desired form, with $a_3, b_3 \in R$.

This is not always possible, because the division polynomial $h = \psi_\ell$ need not be irreducible (indeed, if $E[\ell] \subseteq E(\mathbb{F}_q)$ it will split into linear factors), so the the ring R is not necessarily a field and may contain nonzero elements that are not invertible. This might appear to present an obstacle, but it in fact it only helps us if the denominator d of r is not invertible modulo h. In this case d and h must have a non-trivial common factor $g = \gcd(d,h)$ of degree, and we claim that in fact $\deg g < \deg h$. This is clear when the denominator is $d = a_1 - a_2$, since both a_1 and a_2 are reduced modulo h and therefore have degree less than that of h. When the denominator is $d = 2b_1 f$, we note that f and h must be relatively prime, since roots of f correspond to points of order 2, while roots of h correspond to points of odd order ℓ; therefore g must be a factor of b_1, which has degree less than that of h.

Having found a proper factor g of h, our strategy is to simply replace h by g and restart the computation of t_ℓ. The roots of g correspond to the x-coordinates of a non-empty subset of the affine points in $E[\ell]$ (in fact, a subgroup, as explained below), and it follows from Lemma 9.2 that we can restrict our attention to the action of π_ℓ on points in this subset. This allows us to represent elements of $\mathrm{End}(E[\ell])$ using coordinates in the ring $\mathbb{F}_q[x]/(g(x))$ rather than $\mathbb{F}_q[x]/(h(x))$.

9.5 Algorithm to compute the trace of Frobenius modulo ℓ

We now give an algorithm to compute t_ℓ, the trace of Frobenius modulo ℓ.

Algorithm 9.3. Given an elliptic curve $E : y^2 = f(x)$ over $\mathbb{F}_q$ and an odd prime ℓ, compute t_ℓ as follows:

1. Compute the ℓth division polynomial $h = \psi_\ell \in \mathbb{F}_q[x]$ for E.

2. Compute $\pi_\ell = (x^q \bmod h,\ (f^{(q-1)/2} \bmod h)y)$ and $\pi_\ell^2 = \pi_\ell \circ \pi_\ell$.

3. Use scalar multiplication to compute $q_\ell = q_\ell[1]_\ell$, and then compute $\pi_\ell^2 + q_\ell$.
(If a bad denominator arises, replace h by a proper factor g and go to step 2).

4. Compute $0, \pi_l, 2\pi_l, 3\pi_l, \ldots, c\pi_\ell$, until $c\pi_l = \pi_l^2 + q_l$.
(If an bad denominator arises replace h by a proper factor and go to step 2).

5. Return $t_\ell = c$.

Throughout the algorithm, elements of $\operatorname{End}(E[\ell])$ are represented in the form $(a(x), b(x)y)$, with $a, b \in R = \mathbb{F}_q[x]/(h(x))$, and all polynomial operations take place in the ring R.

The correctness of the algorithm follows from equation (1) and Lemma 9.2. The algorithm is guaranteed to find $c\pi_l = \pi_l^2 + q_l$ in step 4 with $c < \ell$, since $c = t_\ell$ works, by (1). Although we may be working modulo a proper factor g of h, every root x_0 of g is a root of h and therefore corresponds to a pair of nonzero points $P = (x_0, \pm y_0) \in E[\ell]$ for which $\pi_\ell^2(P) - c\pi_\ell(P) + q_\ell P = 0$ holds (there is at least one such root, since g has positive degree), and by Lemma 9.2, we must have $c = t_\ell$.

The computation of the division polynomial in step 1 of the algorithm can be efficiently accomplished using the double-and-add approach described in Problem 3 of Problem Set 3. You will analyze the exact complexity of this algorithm in the next problem set, but it should be clear that its running time is polynomial in $n = \log q$ and ℓ, since all the polynomials involved have degrees that are polynomial in ℓ, and there are at most ℓ iterations in step 4. A simple implementation of the algorithm can be found in the Sage worksheet 18.783 Lecture 9: Schoof's Algorithm.sws.

9.6 Factors of the division polynomial

As we saw when running our implementation of Schoof's algorithm in Sage, we do occasionally find a proper factor of the ℓth division polynomial $h(x)$. This is not too surprising, since there is no reason why h should be irreducible, as noted above. But it is worth noting that when the algorithm finds a factor g of h, the polynomial g always appears to have degree $(\ell - 1)/2$. Why is this the case?

Any point $P = (x_0, y_0)$ for which $g(x_0) = 0$ lies both in $E[\ell]$ and in the kernel of some endomorphism α (since x_0 is a root of the denominator of a rational function defining α). The point P is nonzero, so it generates a cyclic group C of order ℓ, and C must lie in the kernel of α (since $\alpha(mP) = m\alpha(P) = 0$ for all m). It follows that g must have at least $(\ell - 1)/2$ roots, one for each pair of nonzero points $(x_i, \pm y_i)$ in C (note that ℓ is odd). On the other hand, if g has any other roots, then there is another point Q that lies in the intersection of $E[\ell]$ and $\ker \alpha$, and then we must have $\ker \alpha = E[\ell]$, since $E[\ell]$ has ℓ-rank 2. But this would imply that every root of h is a root of g, which is not the case, since g is a proper divisor of h and all of h's roots are distinct. So g has exactly $(\ell - 1)/2$ roots. Reducing the polynomials that define our endomorphism modulo g corresponds to working in the subring $\operatorname{End}(C)$ of $\operatorname{End}(E[\ell])$.

Once we have found such a g, note that the algorithm speeds up by a factor of ℓ, since we are working modulo a polynomial of degree $(\ell - 1)/2$ rather than $(\ell^2 - 1)/2$. While we are unlikely to stumble across such a g by chance once ℓ is large, it turns out that in fact such a g does exist for approximately half of the primes ℓ. Not long after Schoof published his result, Noam Elkies found a way to systematically find such factors g of h as polynomials representing the kernels of isogenies, which allows one to speed up Schoof's algorithm quite dramatically. We will learn about Elkies' technique later in the course when we discuss modular polynomials.

9.7 Some historical remarks

As a historical footnote, when Schoof originally developed this algorithm, it was not clear to him that it had any practical use. This is in part because he (and others) were unduly pessimistic about its practical efficiency (efficient computer algebra implementations were not as widely available then as they are now). Even the simple Sage implementation given in the worksheet is already noticeably faster than baby-steps giant-steps for $q \approx 2^{80}$ and can readily handle computations over fields of cryptographic size (it would likely take a day or two for $q \approx 2^{256}$, but this could be substantially improved with a lower-level implementation).

To better motivate his algorithm, Schoof gave an application that is of purely theoretical interest: he showed that it could be used to deterministically compute the square root of an integer x modulo a prime p in time that is polynomial in $\log p$, for a fixed value of x (we will see exactly how this works when we cover the theory of complex multiplication). Previously, no deterministic polynomial time algorithm was known for this problem, unless one assumes the extended Riemann hypothesis. But Schoof's square-root application is really of no practical use; as we have seen, there are fast probabilistic algorithms to compute square roots modulo a prime, and unless the extended Riemann hypothesis is false, there are even deterministic algorithms that are much faster than Schoof's approach.

By contrast, in showing how to compute $\#E(\mathbb{F}_q)$ in polynomial-time, Schoof solved a practically important problem for which the best previously known algorithms were fully exponential (including the fastest probabilistic approaches), despite the efforts of many working in the field. While perhaps not fully appreciated at the time, this has to be regarded as a major breakthrough, both from a theoretical and practical perspective. Improved versions of Schoof's algorithm (the Schoof-Elkies-Atkin or SEA algorithm) are now the method of choice for computing $\#E(\mathbb{F}_q)$ in fields of large characteristic and are widely used. In particular, the PARI library that is used by Sage for point-counting includes an implementation of the SEA algorithm, and over 256-bit fields it typically takes less than five seconds to compute $\#E(\mathbb{F}_q)$. Today it is feasible to compute $\#(\mathbb{F}_q)$ even when q is a prime with 5,000 decimal digits (over 16,000 bits), which represents the current record.

9.8 The discrete logarithm problem

We now turn to a problem that is generally believed *not* to have a polynomial-time solution. In its most standard form, the *discrete logarithm problem* in a finite group G can be stated as follows:

> *Given $\alpha \in G$ and $\beta \in \langle\alpha\rangle$, find the least positive integer x such that $\alpha^x = \beta$.*

In additive notation (which we will often use), we want $x\alpha = \beta$. In either case, we call x the discrete logarithm of β with respect to the base α and denote it $\log_\alpha \beta$.[4] Note that in the form stated above, where x is required to be positive, the discrete logarithm problem includes the problem of computing the order of α as a special case: $|\alpha| = \log_\alpha 1_G$.

We can also formulate a slightly stronger version of the problem:

> *Given $\alpha, \beta \in G$, compute $\log_\alpha \beta$ if $\beta \in \langle\alpha\rangle$ and otherwise report that $\beta \notin \langle\alpha\rangle$.*

This can be a significantly harder problem. For example, if we are using a Las Vegas algorithm, when β lies in $\langle\alpha\rangle$ we are guaranteed to eventually find $\log_\alpha \beta$, but if not, we will never find it and it may be impossible to tell whether we are just very unlucky or $\beta \notin \langle\alpha\rangle$. On the other hand, with a deterministic algorithm such as the baby-steps giant-steps method, we can unequivocally determine whether β lies in $\langle\alpha\rangle$ or not.

There is also a generalization called the *extended discrete logarithm*:

[4] The multiplicative terminology stems from the fast that most of the early work on computing discrete logarithms was focused on the case where G is the multiplicative group of a finite field.

> *Given $\alpha, \beta \in G$, determine the least positive integer y such that $\beta^y \in \langle\alpha\rangle$, and then output the pair (x, y), where $x = \log_\alpha \beta^y$.*

This yields positive integers x and y satisfying $\beta^y = \alpha^x$, where we minimize y first and x second. Note that there is always a solution: in the worst case $x = |\alpha|$ and $y = |\beta|$.

Finally, one can also consider a *vector form of the discrete logarithm problem*:

> *Given $\alpha_1, \ldots \alpha_r \in G$ and $n_1, \ldots, n_r \in \mathbb{Z}$ such that every $\beta \in G$ can be written uniquely as $\beta = \alpha_1^{e_1} \cdots \alpha_r^{e_r}$ with $e_i \in [1, n_i]$, compute the exponent vector $(e_1, \ldots, e_r)$ associated to a given β.*

Note that the group G need not be abelian in order for the hypothesis to apply, it suffices for G to by *polycyclic* (meaning that it admits a subnormal series with cyclic quotients).

The extended discrete and vector forms of the discrete logarithm problem play an important role in algorithms to compute the structure of a finite abelian group, but in the lectures we will focus primarily on the standard form of the discrete logarithm problem (which we may abbreviate to DLP).

Example 9.4. Suppose $G = \mathbb{F}_{101}^\times$. Then $\log_3 37 = 24$, since $3^{24} \equiv 37 \bmod 101$.

Example 9.5. Suppose $G = \mathbb{F}_{101}^+$. Then $\log_3 37 = 46$, since $46 \cdot 3 = 37 \bmod 101$.

Both of these examples involve groups where the discrete logarithm is easy to compute (and not just because 101 is a small number), but for very different reasons. In Example 9.4 we are working in a group of order $100 = 2^2 \cdot 5^2$. As we will see in the next lecture, when the group order is a product of small primes (i.e. *smooth*), it is easy to compute discrete logarithms. In Example 9.5 we are working in a group of order 101, which is prime, and in terms of the group structure, this represents the hardest case. But in fact it is *very easy* to compute discrete logarithms in the additive group of a finite field! All we need to do is compute the multiplicative inverse of 3 modulo 101 (which is 34) and multiply by 37. This is a small example, but even if the field size is very large, we can use the extended Euclidean algorithm to compute multiplicative inverses in quasi-linear time.

So while the DLP is generally considered a "hard problem", it really depends on which group we are talking about. In the case of the additive group of a finite field the problem is easier not because the group itself is different in any group-theoretic sense, but because it is embedded in a field. This allows us to perform operations on group elements beyond the standard group operation, which is in some sense "cheating".

Even when working in the multiplicative group of a finite field, where the DLP is believed to be much harder, we can do substantially better than in a generic setting. As we shall see, there are sub-exponential time algorithms for this problem, whereas in the generic setting defined below, only exponential time algorithms exist, as we will prove in the next lecture.

9.9 Generic group algorithms

In order to formalize the notion of "not cheating", we define a *generic group algorithm* (or just a *generic algorithm*) to be one that interacts with the group G using a *black box* (sometimes called an *oracle*), via the following interface (think of a black box with four buttons, two input slots, and one output slot):

1. `identity`: output the identity element.
2. `inverse`: given input α, output α^{-1} (or $-\alpha$, in additive notation).

3. `composition`: given inputs α and β, output $\alpha\beta$ (or $\alpha+\beta$, in additive notation).
4. `random`: output a uniformly distributed random element $\alpha \in G$.

All group elements are represented as bit-strings of length $m = O(\log |G|)$ chosen by the black box. Some models for generic group algorithms also include a black box operation for testing equality of group elements, but we will instead assume that group elements are *uniquely identified*; this means that there is an injective identification map $\mathrm{id}\colon G \to \{0,1\}^m$ such that all inputs and outputs to the black box are bit-strings in $\{0,1\}^m$ that correspond to images of group elements under this map. With uniquely identified group elements we can test equality by comparing bit-strings, which does not involve the black box.[5]

The black box may use *any* identification map (including one chosen at random). A generic algorithm is not considered correct unless it works no matter what the identification map is. We have already seen several examples of generic group algorithms, including exponentiation algorithms, fast order algorithms, and the baby-steps giant-steps method.

We measure the time complexity of a generic group algorithm by counting *group operations*, the number of interactions with the black box. This metric has the virtue of being independent of the actual software and hardware implementation, allowing one to make comparisons the remain valid even as technology improves. But if we want to get a complete measure of the complexity of solving a problem in a particular group, we need to multiply the group operation count by the bit-complexity of each group operation, which of course depends on the black box. To measure the space complexity, we count the total number of group identifiers stored at any one time (i.e. the maximum number of group identifiers the algorithm ever has to remember).

These complexity metrics do not account for any other work done by the algorithm. If the algorithm wants to compute a trillion digits of pi, or factor some huge integer, it can effectively do that “for free”. But the implicit assumption is that the cost of any auxiliary computation is at worst proportional to the number of group operations — this is true of all the algorithms we will consider.

References

[1] Joachim von zur Gathen and Jürgen Garhard, *Modern Computer Algebra*, third edition, Cambridge University Press, 2013.

[2] Takakazu Satoh, *On p-adic point counting algorithms for elliptic curves over finite fields*, ANTS V, LNCS **2369** (2002), 43–66.

[3] René Schoof, *Elliptic curves over finite fields and the computation of square roots mod p*. Mathematics of Computation **44** (1985), 483–495.

[4] René Schoof, *Counting points on elliptic curves over finite fields*, Journal de Théorie des Nombres de Bordeaux **7** (1995), 219–254.

[5] Andrew V. Sutherland, *On the evaluation of modular polynomials*, in Proceedings of the Tenth Algorithmic Number Theory Symposium (ANTS X), Open Book Series **1**, Mathematical Science Publishers, 2013, 531–555.

[5] We can also sort bit-strings or index them with a hash table or other data structure; this is essential to an efficient implementation of the baby-steps giant-steps algorithm.