**18.783 Elliptic Curves** **Spring 2013**
**Lecture #10** **03/12/2013**

## 10.1 The discrete logarithm problem

In its most standard form, the *discrete logarithm problem* (DLP) is 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, we want $x\alpha = \beta$. In any case, we call $x$ the discrete logarithm of $\beta$ with respect to the base $\alpha$, denoted by $\log_\alpha \beta$.[1]

We can formulate a slightly stronger version of the problem:

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

This can be a significantly harder problem. For example, say we are using a randomized (Las Vegas) algorithm. If $\beta$ lies in $\langle\alpha\rangle$ then 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 $\beta$ lies in $\langle\alpha\rangle$ or not.

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

> *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 $y = |\beta|$.

**Example 10.1.** Suppose $G = \mathbb{F}_{101}^\times$. Then $\log_3 37 = 24$, since $3^{24} \equiv 37 \bmod 101$.

**Example 10.2.** 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 10.1 we are working in a group of order $100 = 2^2 \cdot 5^2$. As we will see in this lecture, when the group order is a product of small primes (i.e. *smooth*), it is easy to compute discrete logarithms. In Example 10.2 we are working in a group of order 101, which is prime. In terms of the group order, this represents the hardest possible 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 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 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

[1] We will primarily use additive notation, but stick with the multiplicative terminology (hence the word *logarithm*), which is traditional; most of the early work on discrete logarithms was done in the multiplicative group of a finite field.

standard group operation, which is in some sense "cheating". Note that if the Euclidean algorithm were only allowed to use addition and subtraction it would take exponential time.

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 sub-exponential time algorithms for this problem, whereas in the generic case only exponential time algorithms exist, as we will prove in the next lecture. The reason we can do better than in the generic case again arises from the fact that the group in question is embedded in a larger algebraic structure.

## 10.2 Generic group algorithms

To make formal 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* (also 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. `invert`: given input $\alpha$, output $\alpha^{-1}$ (or $-\alpha$, in additive notation).
3. `composition`: given inputs $\alpha$ and $\beta$, output $\alpha\beta$ (or $\alpha + \beta$, in additive notation).
4. `random`: output a uniformly distributed random element $\alpha \in G$.

All group elements processed by the black box are represented as bit-strings of length $k = O(\log |G|)$. 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 $G \to \{0,1\}^k$ such that all inputs and outputs to the black box are bit-strings in $\{0,1\}^k$ 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.[2]

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, which is true of all the algorithms we will consider.

[2]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.

## 10.3 Generic algorithms for the discrete logarithm problem

We now consider generic algorithms for the discrete logarithm problem. We shall assume throughout that $N = |\alpha|$ is known. This is a reasonable assumption, since there is a generic algorithm to compute $N$ using $o(\sqrt{N})$ group operations [5], which is strictly less than the complexity of any of the generic algorithms we consider (at least in the worst case, when $N$ is prime). Indeed, we will prove an $\Omega(\sqrt{N})$ lower bound for prime $N$ in the next lecture.

The cyclic group $\langle\alpha\rangle$ is isomorphic to the additive group $\mathbb{Z}/N\mathbb{Z}$, where $N$ is the order of $\alpha$. In the context of generic group algorithms, we may as well assume $\langle\alpha\rangle$ *is* $\mathbb{Z}/N\mathbb{Z}$, generated by $\alpha = 1$, since every cyclic group of order $N$ looks the same when it is hidden in a black box. Of course with the black box picking arbitrary group identifiers, we cannot actually tell which integer $x$ in $\mathbb{Z}/N\mathbb{Z}$ corresponds to a particular group element $\beta$; indeed, $x$ is precisely the discrete logarithm of $\beta$ that we wish to compute!

Thus computing discrete logarithms amounts to explicitly computing the isomorphism from $\langle\alpha\rangle$ to $\mathbb{Z}/N\mathbb{Z}$. Computing the isomorphism in the reverse direction is easy: this is just exponentiation. Thus we have (in multiplicative notation):

$$\begin{aligned}\mathbb{Z}/N\mathbb{Z} &\simeq \langle\alpha\rangle\\ x &\to \alpha^x\\ \log_\alpha\beta &\leftarrow \beta\end{aligned}$$

Cryptographic applications of the discrete logarithm problem rely on the fact that it is easy to compute $\beta = \alpha^x$ but hard (in general) to compute $x = \log_\alpha \beta$.

### 10.3.1 Linear search

Starting form $\alpha$, compute

$$\alpha, 2\alpha, 3\alpha, \ldots, m\alpha = \beta,$$

and then output $m$ (or if we reach $m > N$, report that $\beta \notin \langle\alpha\rangle$). This uses at most $N$ group operations, and the average over all inputs is $N/2$ group operations.

We mention this algorithm only for the sake of comparison. Its time complexity is not attractive, but it is very space-efficient: it only needs to store one group element, in addition to the inputs $\alpha$ and $\beta$.

### 10.3.2 Baby-steps giant-steps

Pick positive integers $r$ and $s$ such that $rs > N$, and then compute:

$$\begin{array}{ll}\text{baby steps:} & 0,\ \alpha,\ 2\alpha,\ 3\alpha,\ \ldots,\ (r-1)\alpha,\\ \text{giant steps:} & \beta,\ \beta - r\alpha,\ \beta - 2r\alpha,\ \ldots,\ \beta - (s-1)r\alpha,\end{array}$$

A collision occurs when we find a baby step that is equal to a giant step. We then have

$$i\alpha = \beta - jr\alpha,$$

for some nonnegative integers $i < r$ and $j < s$. If $i = j = 0$, then $\beta$ is the identity and $\log_\alpha \beta = N$. Otherwise,

$$\log_\alpha \beta = i + jr.$$

Typically the baby steps are stored in a lookup table, allowing us to check for a collision as each giant step is computed, so we don't necessarily need to compute all the giant steps.

We can easily detect $\beta \notin \langle\alpha\rangle$, since every integer in $[1, N]$ can be written in the form $i + jr$ with $0 \le i < r$ and $0 \le j < s$. If we do not find a collision, then $\beta \notin \langle\alpha\rangle$.

The baby-steps giant-steps algorithm uses $r + s$ group operations, which is $O(\sqrt{N})$ if we choose $r \approx s$. It requires space for $r$ group elements (the baby steps), which is also $O(\sqrt{N})$ but can be made smaller if are willing to increase the running time by making $s$ larger. So there is a trade-off between time and space.

Both of the algorithms above are insensitive to any special properties of $N$, their complexities depend only on its approximate size. In fact we do not even need to know $N$ exactly, we can easily make do with an upper bound. For the next algorithm we consider it is quite important to know $N$ exactly, because our first step will be to factor it.

### 10.3.3 Pohlig-Hellman (and Silver)

Suppose $N = N_1N_2$, where $N_1 \perp N_2$. Then $\mathbb{Z}/N\mathbb{Z} \simeq \mathbb{Z}/N_1\mathbb{Z} \oplus \mathbb{Z}/N_2\mathbb{Z}$, by the Chinese remainder theorem, and we can make this isomorphism completely explicit:

$$\begin{array}{rcr} x & \rightarrow & (x \bmod N_1,\ x \bmod N_2), \\ (M_1x_1 + M_2x_2) \bmod N & \leftarrow & (x_1,\ x_2), \end{array}$$

where

$$M_1 = N_2(N_2^{-1} \bmod N_1) \equiv \begin{cases} 1 \bmod N_1, \\ 0 \bmod N_2, \end{cases} \tag{1}$$

$$M_2 = N_1(N_1^{-1} \bmod N_2) \equiv \begin{cases} 0 \bmod N_1, \\ 1 \bmod N_2. \end{cases} \tag{2}$$

Note that computing $M_i$ and $N_i$ does not involve group operations and is independent of $\beta$.

Now let us consider the computation of $x = \log_\alpha \beta$. Let

$$x_1 = x \bmod N_1 \qquad \text{and} \qquad x_2 = x \bmod N_2,$$

so that $x = M_1x_1 + M_2x_2$, and

$$\beta = (M_1x_1 + M_2x_2)\alpha.$$

Multiplying both sides by $N_2$ and distributing yields

$$N_2\beta = M_1x_1N_2\alpha + M_2x_2N_2\alpha. \tag{3}$$

As you proved in Problem Set 1, the order of $N_2\alpha$ is $N_1$. From (1) and (2) we note that $M_1 \equiv 1 \bmod N_1$ and $M_2 \equiv 0 \bmod N_1$, so (3) can be simplified to

$$N_2\beta = x_1N_2\alpha.$$

We similarly find that $N_1\beta = x_2N_1\alpha$, and therefore

$$x_1 = \log_{N_2\alpha} N_2\beta,$$
$$x_2 = \log_{N_1\alpha} N_1\beta.$$

Assuming we have computed $x_1$ and $x_2$, we may then compute $x = (M_1x1 + M_2x_2) \bmod N$. Thus we have reduced the computation of the discrete logarithm $\log_\alpha \beta$ to the computation of the discrete logarithms $\log_{N_2\alpha} N_2\beta$ and $\log_{N_1\alpha} N_1\beta$. If $N$ is prime this doesn't help (either $N_1 = N$ or $N_2 = N$), but otherwise these two discrete logarithms have bases with smaller orders. In the best case $N_1 \approx N_2$, and we have reduced our original problem to two subproblems of half the size.

By applying the reduction above recursively, we can reduce this to the case where $N$ is a prime power $p^e$, which we now assume. Let $e_0 = \lceil e/2 \rceil$ and $e_1 = \lfloor e/2 \rfloor$. We may write $x = \log_\alpha \beta$ in the form $x = x_0 + p^{e_0}x_1$, with $0 \le x_0 < p^{e_0}$ and $0 \le x_1 < p^{e_1}$. We then have

$$\begin{aligned} \beta &= (x_0 + p^{e_0}x_1)\alpha, \\ p^{e_1}\beta &= x_0 p^{e_1}\alpha + x_1 p^e \alpha, \end{aligned}$$

but the second term is zero, because $\alpha$ has order $N = p^e$, so

$$x_0 = \log_{p^{e_1}\alpha} p^{e_1}\beta.$$

We also have $\beta - x_0\alpha = p^{e_0}x_1\alpha$, thus

$$x_1 = \log_{p^{e_0}\alpha}(\beta - x_0\alpha).$$

If $N$ is not prime, this again reduces the computation of $\log_\alpha \beta$ to the computation of two smaller discrete logarithms (of roughly equal size).

The Pohlig-Hellman method [2] recursively applies the two reductions above to reduce the problem to a set of discrete logarithm computations in groups of prime order.[3] For these computations we must revert to some other method, such as baby-steps giant-steps (or Pollard-rho, which we will see shortly). When $N$ is a prime $p$, the complexity is then $O(\sqrt{p})$ group operations.

### 10.3.4 Complexity analysis

Suppose $N$ is a product of relatively prime powers, $q_1 \dots q_k$. Reducing to the prime-power case involves at most $\lg k = O(\log n)$ levels of recursion, where $n = \log N$. The largest possible exponent of a prime power is $\lg N = O(n)$, thus reducing prime powers to the prime case involves at most an additional $O(\log n)$ levels of recursion.

The total depth of the recursion tree is thus $O(\log n)$, and note that the product of the orders of the bases used at any given level of the recursion tree is equal to $N$. The number of group operations required at each internal node of the recursion tree is linear in the order of the base, thus if we exclude the primes cases at the leaves, every layer of the recursion tree uses $O(n)$ group operations. Each leaf requires $O(\sqrt{p_i})$ group operations, so the total running time is

$$O\left(n \log n + \sum e_i \sqrt{p_i}\right)$$

group operations, which we may bound by

$$O\left(n \log n + n\sqrt{p}\right),$$

where $p$ is the largest prime dividing $N$. The space complexity is $O(\sqrt{p})$ group elements, assuming we use a baby-steps giant-steps search for the prime cases; this can be reduced to

[3]The original algorithm of Pohlig and Hellman actually used an iterative approach that is not as fast as the recursive approach suggested here is faster (a recursive approach for the prime-power case is given by Shoup in [4, §11.2.3]).

$O(1)$ using the Pollard-rho method (which is the next algorithm we will consider), but this results in a probabilistic (Las Vegas) algorithm, whereas the basic Pohlig-Hellman approach is completely deterministic.

The Pohlig-Hellman algorithm can be extremely efficient when $N$ is composite; if $N$ is sufficiently smooth it runs in quasi-linear time (computing discreate logarithms in a group of order $10^{1000}$ takes almost no time at all). This underscores the importance of using groups of prime or near-prime order in any cryptographic application based on the discrete logarithm problem. This is one of the main motivations for efficient point-counting algorithms for elliptic curves: we really want to know exactly what the group order is.

## 10.4 Randomized algorithms for the discrete logarithm problem

So far we have only considered deterministic algorithms. We will not achieve a better time complexity bound with randomization, but we can achieve a much better space complexity; this also makes it much easier to parallelize the algorithm, which is crucial for large-scale computations (one can construct a parallel version of the baby-steps giant-steps algorithm, but detecting collisions is more complicated and requires a lot of communication).

### 10.4.1 The birthday paradox

Recall what the *birthday paradox* tells us about collision frequency: if we drop $\Omega(\sqrt{N})$ balls randomly into $O(N)$ bins then the probability that some bin contains more than one ball is bounded below by some nonzero constant that we can make arbitrarily close to 1. Given $\beta \in \langle\alpha\rangle$, the baby-steps giant-steps method for computing $\log_\alpha \beta$ can be viewed as dropping $\sqrt{2N}$ balls that are linear combinations of $\alpha$ and $\beta$ (the baby steps are linear combinations of $\alpha$ alone) into $N$ bins corresponding to the elementd of $\langle\alpha\rangle$. Of course these balls are not dropped randomly, they are dropped in a pattern that guarantees a collision.

But if we instead just computed $\sqrt{2N}$ linear combinations of $\alpha$ and $\beta$ at random, the birthday paradox tells us that we would still have a good chance of finding a collision (better than 50/50, in fact). The main problem with this approach is that in order to find the collision we might need to keep track of all the linear combinations we have computed, which would take a lot of space. In order to take advantage of the birthday paradox in a way that uses less space we need to be a bit more clever.

### 10.4.2 Random walks on a graph

Let us now view the group $\langle\alpha\rangle$ as the vertext set $V$ of a graph. Suppose we use a random function $f\colon V \to V$ to construct a walk from a random starting point $v_0$ as follows:

$$\begin{aligned} v_1 &= f(v_0)\\ v_2 &= f(v_1)\\ v_3 &= f(v_2)\\ &\vdots \end{aligned}$$

Since $f$ is a function, if we ever repeat a vertex, say $v_\rho = v_\lambda$ for some $\rho > \lambda$, we will be permanently stuck in a cycle, since we then have $f(v_{\rho+i}) = f(v_{\lambda+i})$ for all $i \geq 0$. This is a good thing, because once we are in this cycle *every* step corresponds to a collision (a pair of vertices with different indices in our random walk that correspond to the same group element). Note that $V$ is finite, so we must eventually hit a cycle.

Our random walk thus consists of two parts, a path from $v_0$ to the vertex $v_\lambda$, the first vertex that is visited more than once, and a cycle consisting of the vertices $v_\lambda, v_{\lambda+1}, \ldots, v_{\rho-1}$. The can be visualized as a path in the shape of the Greek letter $\rho$, which explains the name of the *$\rho$-method* we wish to consider.

A collision doesn't necessarily tell us anything on its own, but if we augment the function $f$ appropriately, it will. We will construct $f$ so that the vertices in our random walk correspond to linear combinations $a\alpha + b\beta$ whose coefficients $a$ and $b$ we know. But let us first compute the expected number of steps a random walk takes to reach its first collision.

**Theorem 10.3.** *Let $V$ be a finite set. For any $v_0 \in V$, the expected value of $\rho$ for a walk from $v_0$ defined by a random function $f\colon V \to V$ is*

$$\mathrm{E}[\rho] \sim \sqrt{\pi N/2},$$

*as the cardinality $N$ of $V$ tends to infinity.*

This theorem was stated in lecture without proof; we now give an elementary proof.

*Proof.* Let $P_n = \Pr[\rho > n]$. We have $P_0 = 1$ and $P_1 = (1 - 1/N)$, and in general

$$P_n = \left(1 - \frac{1}{N}\right)\left(1 - \frac{2}{N}\right)\cdots\left(1 - \frac{n}{N}\right) = \prod_{i=1}^{n}\left(1 - \frac{i}{N}\right)$$

for any $n < N$ (and $P_n = 0$ for $n \geq N$). We compute the expectation of $\rho$ as

$$\begin{aligned}
\mathrm{E}[\rho] &= \sum_{n=1}^{N-1} n \cdot \Pr[\rho = n] \\
&= \sum_{n=1}^{N-1} n \cdot (P_{n-1} - P_n), \\
&= 1(P_0 - P_1) + 2(P_1 - P_2) + \ldots + n(P_{n-1} - P_n) \\
&= \sum_{n=0}^{N-1} P_n - nP_n. \qquad (4)
\end{aligned}$$

In order to determine the asymptotic behavior of $E[\rho]$ we need tight bounds on $P_n$. Using the fact that $\log(1-x) < -x$ for $0 < x < 1$, we obtain an upper bound on $P_n$:

$$\begin{aligned}
P_n &= \exp\left(\sum_{i=1}^{n} \log\left(1 - \frac{i}{N}\right)\right) \\
&< \exp\left(-\frac{1}{N}\sum_{i=1}^{n} i\right) \\
&< \exp\left(\frac{-n^2}{2N}\right).
\end{aligned}$$

To establish a lower bound, we use the fact that $\log(1-x) > -x - x^2$ for $0 < x < \frac{1}{2}$, which can be verified using the Taylor series expansion for $\log(1-x)$.

$$\begin{aligned} P_n &= \exp\left(\sum_{i=1}^{n} \log\Big(1 - \frac{i}{N}\Big)\right) \\ &> \exp\left(-\sum_{i=1}^{n} \Big(\frac{i}{N} + \frac{i^2}{N^2}\Big)\right). \end{aligned}$$

We now let $M = N^{3/5}$ and assume $n < M$. In this range we have

$$\begin{aligned} \sum_{i=1}^{n} \Big(\frac{i}{N} + \frac{i^2}{N^2}\Big) &< \sum_{i=1}^{n} \Big(\frac{i}{N} + N^{-\frac{4}{5}}\Big) \\ &< \frac{n^2+n}{2N} + N^{-\frac{1}{5}} \\ &< \frac{n^2}{2N} + \frac{1}{2}N^{-\frac{2}{5}} + N^{-\frac{1}{5}} \\ &< \frac{n^2}{2N} + 2N^{-\frac{1}{5}}, \end{aligned}$$

which implies

$$\begin{aligned} P_n &> \exp\Big(\frac{-n^2}{2N}\Big) \exp\Big(-2N^{-\frac{1}{5}}\Big) \\ &= \big(1 + o(1)\big) \exp\Big(\frac{-n^2}{2N}\Big). \end{aligned}$$

We now return to the computation of $\mathrm{E}[\rho]$. From (4) we have

$$\mathrm{E}[\rho] = \sum_{n=0}^{\lfloor M \rfloor} P_n + \sum_{n=\lceil M \rceil}^{N-1} P_n + o(1) \tag{5}$$

where the error term comes from $nP_n < n\exp\big(\frac{-n^2}{2N}\big) = o(1)$ (we use $o(1)$ to denote any term whose absolute value tends to 0 as $N \to \infty$). The second sum is negligible, since

$$\begin{aligned} \sum_{n=\lceil M \rceil}^{N-1} P_n &< N\exp\Big(-\frac{M^2}{2N}\Big) \\ &= N\exp\Big(-\frac{1}{2}N^{-\frac{1}{5}}\Big) \\ &= o(1). \end{aligned} \tag{6}$$

For the first sum we have

$$\begin{aligned}
\sum_{n=0}^{\lceil M\rceil} P_n &= \sum_{n=0}^{\lceil M\rceil} \big(1+o(1)\big)\exp\Big(-\frac{n^2}{2N}\Big)\\
&= (1+o(1))\int_0^\infty e^{-\frac{x^2}{2N}}\,dx + O(1)\\
&= (1+o(1))\sqrt{2N}\int_0^\infty e^{u^2}\,du + O(1)\\
&= (1+o(1))\sqrt{2N}(\sqrt{\pi}/2)\\
&= (1+o(1))\sqrt{\pi N/2}. \qquad (7)
\end{aligned}$$

Plugging (6) and (7) in to (5) yields the desired result. □

**Remark 10.4.** One can also show $\mathrm{E}[\lambda] = E[\sigma] = \frac{1}{2}\mathrm{E}[\rho] = \sqrt{\pi N/8}$, where $\sigma = \rho - \lambda$ is the length of the cycle.

In the baby-steps giant-steps algorithm (BSGS), if we assume that the discrete logarithm is uniformly distributed over $[1, N]$, then we should use $\sqrt{N/2}$ baby steps and expect to find the discrete logarithm after $\sqrt{N/2}$ giant steps, on average, using a total of $\sqrt{2N}$ group operations. But note that $\sqrt{\pi/2} \approx 1.25$ is less than $\sqrt{2} \approx 1.41$, so we may hope to compute discrete logarithms faster than BSGS (on average) by simulating a random walk. Of course the worst-case running time for BSGS is better, we will never need more than $\sqrt{2N}$ giant steps, but with a random walk the (astronomically unlikely) worst case is $N$ steps.

## 10.5 Pollard-$\rho$ Algorithm

We now present the Pollard-$\rho$ algorithm for computing $\log_\alpha \beta$. As noted earlier, a collision in a random walk is useful to us only if we know how to express the colliding group elements as (independent) linear combinations of $\alpha$ and $\beta$. Thus we extend the function $f\colon G \to G$ that defines our random walk to a function

$$f\colon (\mathbb{Z}/N\mathbb{Z})^2 \times G \to (\mathbb{Z}/N\mathbb{Z})^2 \times G,$$

which, given $(a, b, \gamma)$ such that $\alpha^a\beta^b = \gamma$, outputs some $(a', b', \gamma')$ such that $\alpha^{a'}\beta^{b'} = \gamma'$.

There are several ways to define such a function $f$, one of which is the following. First fix $r$ distinct group elements $\delta_i = c_i\alpha + d_i\beta$ for some randomly chosen $c_i, d_i \in \mathbb{Z}/N\mathbb{Z}$. In order to simulate a random walk, we don't want $r$ to be too small: empirically a value of around 20 works well [6]. Then define $f(a, b, \gamma) = (a + c_i, b + d_i, \gamma + \delta_i)$ where $i = h(\gamma)$ is determined by a randomly chosen *hash function*

$$h\colon G \to \{1, \ldots, r\}.$$

In practice we don't choose $h$ randomly, we just need the preimages $h^{-1}(i)$ to partition $G$ into $r$ subsets of roughly equal size (for example, we might just reduce the integer corresponding to the bit-string that uniquely represents $\gamma$ modulo $r$, assuming that these integers are roughly equidistributed mod $r$).[4]

[4]Note the importance of unique identifiers. We must be sure that $\gamma$ is always hashed to to the same value. Using a non-unique representation such as projective points on an elliptic curve will not achieve this.

To start our random walk, we pick random $a_0, b_0 \in \mathbb{Z}/N\mathbb{Z}$ and let $\gamma_0 = \alpha^{a_0}\beta^{b_0}$. The walk defined by the iteration function $f$ is then known as an $r$-*adding* walk. Note that if $(a_{j+1}, b_{j+1}, \gamma_{j+1}) = f(a_j, b_j, \gamma_j)$, the value of $\gamma_{j+1}$ depends only on $\gamma_j$, not on $a_j$ or $b_j$, so $f$ does define a walk in the same sense as before. We now give the algorithm.

**Algorithm 10.5** (Pollard-$\rho$)**.**

1. Compute $\delta_i = c_i\alpha + d_i\beta$ for $r$ randomly chosen values of $c_i, d_i \in \mathbb{Z}/N\mathbb{Z}$.
2. Compute $\gamma_0 = a_0\alpha + b_o\beta$ for randomly chosen $a_0, b_0 \in \mathbb{Z}/N\mathbb{Z}$.
3. Compute $(a_j, b_j, \gamma_j) = f(a_{j-1}, b_{j-1}, \gamma_{j-1})$ for $j = 1, 2, 3, \ldots$, until we find a collision, i.e. $\gamma_k = \gamma_j$ for some $k > j$.
4. The collision $\gamma_k = \gamma_j$ implies $a_j\alpha + b_j\beta = a_k\alpha + b_k\beta$. Provided that $b_k - b_j$ is invertible in $\mathbb{Z}/N\mathbb{Z}$, we return $\log_\alpha \beta = \frac{a_j - a_k}{b_k - b_j} \in \mathbb{Z}/N\mathbb{Z}$, otherwise we simply start over.

Note that if $N = |\alpha|$ is a large prime, it is extremely likely that $b_k - b_j$ will be invertible. In any case, by restarting we ensure that the algorithm is guaranteed to eventually terminate (with probability 1), since it is certainly possible to have $\gamma_0 = \alpha^x$ and $\gamma_1 = \beta$, for example. With this implementation the Pollard rho algorithm is a Las Vegas algorithm, not a Monte Carlo algorithm as it is often described in the literature.

The description above does not specify exactly how we should detect collisions. A simple method is to store all the $\gamma_j$ as they are computed and look for a collision during each iteration. However, this implies a space complexity of $\rho$, which we expect to be on the order of $\sqrt{N}$. But we can use dramatically less space than this.

The key point is that once the walk enters a cycle, it will remain inside this cycle forever, and *every* step inside the cycle produces a collision. Therefore it is not necessary to detect a collision at the exact moment we enter the cycle, we can afford a slight delay. We now consider two space-efficient methods for doing this.

## 10.6 Floyd's cycle detection method

Floyd's cycle detection method [1, p. 4] minimizes the space required: it keeps track of just two triples $(a_j, b_j\gamma_j)$ and $(a_k, b_k, \gamma_k)$ that correspond to vertices of the walk (of course it also needs to store $c_i, d_i, \gamma_i$ for $0 \le i < r$). The method is typically described in terms of a tortoise and a hare that are both traveling along the walk. They both start at the same point $\gamma_0$, but in each iteration the hare takes two steps, while the tortoise takes just one. Thus in step 3 of Algorithm 10.5 we compute

$$\begin{aligned}(a_j, b_j, \gamma_j) &= f(a_{j-1}, b_{j-1}, \gamma_{j-1})\\ (a_k, b_k, \gamma_k) &= f(f(a_{k-1}, b_{k-1}, \gamma_{k-1})).\end{aligned}$$

The triple $(a_j, b_j\gamma_j)$ corresponds to the tortoise, and the triple $(a_k, b_k, \gamma_k)$ corresponds to the hare. Once the tortoise enters the cycle, the hare (who must already be in the cycle) is guaranteed to collide with the tortoise within $\sigma/2$ iterations, where $\sigma$ is the length of the cycle (to see this, note that the hare cannot pass the tortoise without landing on it). On average, we expect it to take $\frac{\sigma}{4}$ iterations for the hare to catch the tortoise and produce a collision, which we detect by testing whether $\gamma_j = \gamma_k$ after each iteration.

The expected number of iterations is thus $\mathrm{E}[\lambda + \sigma/4] = 5/8\mathrm{E}[\rho]$. But notice that each iteration now requires three group operations, so the algorithm is actually slower by a factor of 15/8. Still, this achieves a time complexity of $O(\sqrt{N})$ group operations while storing just $O(1)$ group elements, which is a dramatic improvement.

## 10.7 The method of distinguished points

The "distinguished points" method (attributed to Rivest) uses more space, say $O(\log^c N)$ group elements, for some constant $c$, but it detects cycles in essentially optimal time (within a factor of $1 + o(1)$ of the best possible), and uses just one group operation per iteration.

The idea is to "distinguish" a certain subset of $G$ by fixing a random boolean function $B\colon G \to \{0,1\}$ and calling the elements of $B^{-1}(1)$ *distinguished points*. We don't want the set of distinguished points to be too large, since we have to store them, but we want our walk to contain quite a few distinguished points; ideally we should choose $B$ so that $|B^{-1}(1)| \simeq \frac{\log^c N}{\sqrt{N}}$.

One way to define such a function $B$ is to hash group elements to bit-strings of length $k$ via a hash function $\tilde{h}\colon G \to \{0,1\}^k$, and then let $B(\gamma) = 1$ if and only if $\tilde{h}(\gamma)$ is the zero vector.[5] With $k = \frac{1}{2}\log_2 N - c\log_2 \log N$ we achieve $|B^{-1}(1)|$ as desired. An easy and very efficient way to construct the hash function $\tilde{h}$ is to use the $k$ least significant bits of the bit-string that uniquely represents the group element. For points on elliptic curves, we should use bits from the $x$-coordinate, since this will allow us to detect more general collisions of the form $\gamma_j = \pm\gamma_k$ (we can determine the sign by checking $y$-coordinates).

**Algorithm 10.6** (Pollard rho using distinguished points)**.**

```
1. Pick random $c_i, d_i, a_0, b_0 \in \mathbb{Z}/N\mathbb{Z}$ and compute $\delta_i = c_i\alpha + d_i\beta$ and $\gamma_0 = a_0\alpha + b_0\beta$ as
   in the original Pollard-rho algorithm.
2. Initialize the set $D$ of distinguished points to the empty set.
3. For $j = 1, 2, 3, \ldots$:
   a. Compute $(a_j, b_j, \gamma_j) = f(a_{j-1}, b_{j-1}, \gamma_{j-1})$.
   b. If $B(\gamma_j) = 1$ then
      i. If there exists $(a_k, b_k, \gamma_k) \in D$ with $\gamma_j = \gamma_k$ then return $\log_\alpha \beta = \frac{a_j - a_k}{b_k - b_j}$ if
         $\gcd(b_k - b_j, N) = 1$ and restart otherwise.
      ii. If not, replace $D$ by $D \cup \{(a_j, b_j, \gamma_j)\}$ and continue.
```

A key feature of the distinguished points method is that it is well-suited to a parallel implementation, which is critical for any large-scale discrete logarithm computation. Suppose we have many CPUs all running the same algorithm independently. If we have $\sqrt{N}$ CPUs, then after just one step there is a good chance of a collision, and in general if we have $m$ CPUs we expect to get a collision within $O(\sqrt{N}/m)$ steps. We can detect this collision as soon as the CPUs involved in the collision reach a distinguished point, which we expect to occur within $O(\log^c N)$ steps. However, the individual CPUs cannot realize this themselves, since we do not assume they share the same set $D$ of distinguished points. Instead, whenever a CPU encounters a distinguished point, it sends the corresponding triple to a central server that is responsible for detecting collisions. This scenario is also called a $\lambda$-*search*, since the collision typically occurs between paths with different starting points

[5] Note that $\tilde{h}$ is not the same as the hash function $h\colon G \to \{1, 2, 3, \ldots, r\}$ used in Algorithm 10.5.

that then follow the same trajectory (forming the shape of the letter $\lambda$, rather than the letter $\rho$).

There is one important issue that remains to be addressed. If there are no distinguished points in the cycle then Algorithm 10.6 will never terminate!

The solution is to let the distinguished set $S$ grow with time. We begin with $S = \tilde{h}^{-1}(\mathbf{0})$, where $\tilde{h}\colon G \to \{0,1\}^k$ with $k = \frac{1}{2}\log_2 N - c\log_2 \log N$. Every $\sqrt{\pi N/2}$ iterations, we decrease $k$ by 1. This effectively doubles the number of distinguished points, and when $k$ reaches zero we consider every point to be distinguished. This guarantees termination, and the expected space is still just $O(\log^c N)$ group elements (notice that we are generally only increasing the size of $S$ in situations where $D$ is not growing).

## References

[1] D. E. Knuth, *The Art of Computer Programming, vol. II: Semi-numerical Algorithms*, Addison-Wesley, 1969.

[2] S. C. Pohlig and M. E. Hellman, *An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance*, IEEE Transactions on Information Theory **24** (1978), 106–110.

[3] J. M. Pollard, *Monte Carlo methods for index computation (mod p)*, Mathematics of Computation **143** (1978), 918–924.

[4] V. Shoup, *A Computational Introduction to Number Theory and Algebra*, Cambridge University Press, 2005.

[5] A. V. Sutherland, *Order computations in generic groups*, PhD thesis, Massachusetts Institute of Technology, 2007.

[6] E. Teske, *On random walks for Pollard's rho method*, Mathematics of Computation **70** (2001), 809–825.