10 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 [9], which is strictly less than the complexity of any generic algorithm for this problem (we will prove an $\Omega(\sqrt{N})$ lower bound).

The cyclic group $\langle\alpha\rangle$ is isomorphic to the additive group $\mathbb{Z}/N\mathbb{Z}$. 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 in $\{0,1\}^m$, we cannot actually tell which integer x in $\mathbb{Z}/N\mathbb{Z}$ corresponds to a particular group element β; indeed, x is precisely the discrete logarithm of β that we wish to compute!

Thus computing discrete logarithms amounts to explicitly computing the isomorphism from $\langle\alpha\rangle$ to $\mathbb{Z}/N\mathbb{Z}$ that sends α to 1. Computing the isomorphism in the reverse direction is easy: this is just exponentiation. Thus we have (in multiplicative notation):

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

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 $x = \log_\alpha \beta$.

10.1 Linear search

Starting form α, compute

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

and then output x (or if we reach $N\alpha$, 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 we note that its space complexity is just $O(1)$ group elements.

10.2 Baby-steps giant-steps

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

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

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 β 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 $r \approx s \approx \sqrt{N}$. 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; there is thus a time-space trade-off we can make, but the product of the time and space complexity is always $\Omega(N)$.

The two 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.

10.3 The Pohlig-Hellman algorithm

We now introduce the Pohlig-Hellman[1] algorithm, a recursive method to reduce the discrete logarithm problem in cyclic groups of composite order to discrete logarithm problems in cyclic groups of prime order.

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:

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

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 β.

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

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

[1]The article by Pohlig and Hellman [4] notes that essentially equivalent versions of the algorithm were independently found by Ronald Silver, and later by Richard Schroeppel and H.Block; neither of these earlier discoveries was ever published.

We similarly find that $N_1\beta = x_2 N_1 \alpha$, 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 $x = (M_1 x_1 + M_2 x_2) \bmod N$. 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

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

but the second term is zero, because α 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 [4] recursively applies the two reductions above to reduce the problem to a set of discrete logarithm computations in groups of prime order.[2] 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.

10.4 Complexity analysis

Suppose N is a product of relatively prime powers, $q_1 \dots q_r$. Reducing to the prime-power case involves at most $\lg r = O(\log n)$ levels of recursion, where $n = \log N$ (in fact the Prime Number Theorem implies $\lg r = O(\log n / \log\log n)$, but we don't need this). 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 $O(\log n)$. Note that we do not need to assume anything about the prime factorization of N in order to obtain this bound; in particular, even if the q_i are not approximately equal in size, the bound still holds.

[2] The original algorithm of Pohlig and Hellman actually used an iterative approach that is not as fast as the recursive approach suggested here. The recursive approach for the prime-power case that we use here appears in [7, §11.2.3]. When $N = p^e$ is a power of a prime $p = O(1)$, the complexity of the original Pohlig-Hellman algorithm is $O(n^2)$, versus the $O(n \log n)$ bound we obtain here (this can be further improved to $O(n \log n / \log\log n)$ via [10]).

The product of the orders of the bases used at any given level of the recursion tree is always equal to N. The number of group operations required at each internal node of the recursion tree is linear in the logarithm of the order of the base, since only $O(1)$ scalar multiplications are needed in each recursive step. Thus if we exclude the primes order 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, where the sum is over the distinct prime divisors p_i of N. We can also bound this 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 $O(1)$ 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 its running time is quasi-linear in $n = \log N$, essentially the same as for exponentiation. Thus it is quite important to use groups of prime (or near-prime) order in cryptographic applications of the discrete logarithm problem. This is one of the motivations for efficient point-counting algorithms for elliptic curves: we really need to know the exact group order before we can consider a group suitable for cryptographic use.

10.5 Randomized algorithms for the discrete logarithm problem

So far we have only considered deterministic algorithms for the discrete logarithm problem. We now want to consider probabilistic methods. Randomization will not allow us to achieve a better time complexity (a fact we will prove shortly), 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).

10.5.1 The birthday paradox

Recall what the so-called *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 α and β (the baby steps are linear combinations of α 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 α and β at random, 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.

10.5.2 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 $v_0 \in V$ 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_λ, 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 ρ, which explains the name of the *ρ-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 10.1. *Let V be a finite set. For any $v_0 \in V$, the expected value of ρ 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; here 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

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

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

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

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 $\log(1-x)$.

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

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

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

which implies

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

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 $nP_n < n \exp\left(\frac{-n^2}{2N}\right) = o(1)$ (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

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

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

Remark 10.2. One can similarly 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 slightly faster than BSGS (on average) by simulating a random walk. Of course the worst-case running time for BSGS is better, since we will never need more than $\sqrt{2N}$ giant steps, but with a random walk the (very unlikely) worst case is N steps.

10.6 Pollard-ρ Algorithm

We now present the Pollard-ρ 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 α and β. Thus we extend the function $f\colon G \to G$ that defines our random walk to a function

$$f\colon \mathbb{Z}/N\mathbb{Z} \times \mathbb{Z}/N\mathbb{Z} \times G \to \mathbb{Z}/N\mathbb{Z} \times \mathbb{Z}/N\mathbb{Z} \times G,$$

which, given (a, b, γ) such that $a\alpha + b\beta = \gamma$, outputs some (a', b', γ') such that $a'\alpha + b'\beta = \gamma'$.

There are several ways to define such a function f, one of which is the following. We 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 [11]. 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 reduce the integer corresponding to the bit-string that uniquely represents γ modulo r, assuming that these integers are roughly equidistributed mod r).[3]

To start our random walk, we pick random $a_0, b_0 \in \mathbb{Z}/N\mathbb{Z}$ and let $\gamma_0 = a_0\alpha + b_0\beta$. 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 γ_{j+1} depends only on γ_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 10.3 (Pollard-ρ)**.**

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 terminates with probability 1, since it is certainly possible to have $\gamma_0 = x\alpha$ and $\gamma_1 = \beta$, where $x = \log_\alpha \beta$, for example. With this implementation the Pollard rho algorithm is a Las Vegas algorithm, not a Monte Carlo algorithm as it is often referred to in the literature (due to the title of [6]).

The description above does not specify exactly how we should detect collisions. A simple method is to store all the γ_j as they are computed and look for a collision during each iteration. However, this implies a space complexity of ρ, 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.

10.7 Floyd's cycle detection method

Floyd's cycle detection method [3, p. 4] minimizes the space required: it keeps track of just two triples $(a_j, b_j\gamma_j)$ and (a_k, b_k, γ_k) that correspond to vertices of the walk (of course it also needs to store c_i, d_i, γ_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 start with the same γ_0, but in each iteration the hare takes two steps, while the tortoise takes just one. Thus in step 3 of Algorithm 10.3 we compute

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

The triple $(a_j, b_j\gamma_j)$ corresponds to the tortoise, and the triple (a_k, b_k, γ_k) corresponds to the hare. Once the tortoise enters the cycle, the hare (who must already be in the cycle) is

[3]Note the importance of unique identifiers. We must be sure that γ 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.

guaranteed to collide with the tortoise within $\sigma/2$ iterations, where σ 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 $\sigma/4$ 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.

10.8 The method of distinguished points

The "distinguished points" method (attributed to Rivest) uses slightly 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), using 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

$$\#B^{-1}(1) \approx \frac{\log^c N}{\sqrt{N}}.$$

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.[4] If we set $k = \frac{1}{2}\log_2 N - c\log_2\log N$ then $B^{-1}(1)$ will have the desired cardinality. 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 collisions of the form $\gamma_j = \pm\gamma_k$ (we can determine the sign by checking y-coordinates).

Algorithm 10.4 (Pollard-ρ 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 very well-suited to a parallel implementation, which is critical for any large-scale discrete logarithm computation.

[4] Note that $\tilde{h}$ is not the same as the hash function $h\colon G \to \{1, 2, 3, \ldots, r\}$ used in Algorithm 10.3.

Suppose we have many processors all running the same algorithm independently. If we have, say, $\sqrt{N}$ processors, then after just one step there is a good chance of a collision, and in general if we have m processorss we expect to get a collision within $O(\sqrt{N}/m)$ steps. We can detect this collision as soon as the processors involved in the collision reach a distinguished point, which we expect to occur within $O(\log^c N)$ steps. However, the individual processors cannot realize this themselves, since we do not assume they share the same set D of distinguished points. Instead, whenever a processor 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 λ-*search*, since the collision typically occurs between paths with different starting points that then follow the same trajectory (forming the shape of the letter λ, rather than the letter ρ).

There is one important detail that must be addressed: if there are no distinguished points in the cycle then Algorithm 10.4 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, which means that we likely have a very short cycle).

10.9 Current ECDLP records

The most recent record for computing discrete logarithms on elliptic curves over finite fields involves a base point α with 112-bit prime order on an elliptic curve $E/\mathbb{F}_q$ with $q = 2^{113}$. This record was announced just last month [12] and used a parallel Pollard-rho search with some variant of the distinguished points method for collision detection. The computation was run on a 10-core Kintex-7 FPGA cluster and reportedly took about 82 days. The record for elliptic curves over prime fields was set in 2009 and was of similar size (112-bit prime). It also used a parallel Pollard-rho search running on a cluster of 200 PlayStation 3 game consoles and took about 6 months [1].

10.10 A generic lower bound for the discrete logarithm problem

We will now prove an essentially tight lower bound for solving the discrete logarithm problem with a generic group algorithm. We will show that if p is the largest prime divisor of N, then any generic group algorithm for the discrete logarithm problem must use $\Omega(\sqrt{p})$ group operations. In the case that the group order $N = p$ is prime this bound is tight, since we have already seen that the problem can be solved with $O(\sqrt{N})$ group operations using the baby-steps giant-steps method, and the Pohlig-Hellman complexity bound $O(n\log n + n\sqrt{p})$ shows that it is tight in general, up to logarithmic factors.

This lower bound applies not only to deterministic algorithms, but also to randomized algorithms: a generic Monte Carlo algorithm for the discrete logarithm problem must use $\Omega(\sqrt{p})$ group operations in order to be correct with probability greater than $1/2$, and the expected running time of any generic Las Vegas algorithm for the discrete logarithm problem is $\Omega(\sqrt{p})$ group operations.[5]

[5] Any Monte Carlo algorithm for the discrete logarithm problem can be easily converted to a Las Vegas algorithm: use a scalar multiplication to verify the result and rerun the algorithm if it is incorrect.

The following theorem due to Shoup [8] generalizes an earlier result of Nechaev [5]. Our presentation here differs slightly from Shoup's and gives a sharper bound, but the proof is essentially the same. Recall that in our generic group model, each group element is uniquely represented as a bit-string via an injective identification map $\mathrm{id}\colon G \to \{0,1\}^m$, where $m = O(\log |G|)$.

Theorem 10.5 (Shoup)**.** *Let $G = \langle \alpha \rangle$ be group of order N. Let $\mathcal{B}$ be a black box for G supporting the operations* `identity`, `inverse`, *and* `compose`, *using a random identification map $\mathrm{id}\colon G \to \{0,1\}^m$. Let $\mathcal{A}\colon \{0,1\}^m \times \{0,1\}^m \to \mathbb{Z}/N\mathbb{Z}$ be a randomized generic group algorithm that makes at most $s - 4\lceil \log_2 N \rceil$ calls to $\mathcal{B}$, for some integer s, and let x denote a random element of $\mathbb{Z}/N\mathbb{Z}$. Then*

$$\Pr_{x,\mathrm{id},\tau}[\mathcal{A}(\mathrm{id}(\alpha), \mathrm{id}(x\alpha)) = x] < \frac{s^2}{2p},$$

where τ denotes the random coin-flips made by $\mathcal{A}$ and p is the largest prime factor of N.

Note that $\mathcal{A}$ can generate random elements of G by computing $z\alpha$ for random $z \in \mathbb{Z}/N\mathbb{Z}$ (we can assume that $\mathcal{A}$ is given the group order N — this only makes the theorem stronger). The theorem includes deterministic algorithms as the special case where $\mathcal{A}$ does not use any bits of τ. Bounding the number of calls $\mathcal{A}$ makes to $\mathcal{B}$ might appear to preclude Las Vegas algorithms, but we will derive a corollary that addresses this.

Proof. To simply the proof, we will replace $\mathcal{A}$ by an algorithm $\mathcal{A}'$ that does the following:

1. Use $\mathcal{B}$ to compute $\mathrm{id}(N\alpha) = \mathrm{id}(0)$.
2. Simulate $\mathcal{A}$, using $\mathrm{id}(0)$ to replace `identity` operations, to get $y = \mathcal{A}(\mathrm{id}(\alpha), \mathrm{id}(x\alpha))$.
3. Use $\mathcal{B}$ to compute $\mathrm{id}(y\alpha)$.

In the description above we assume that the inputs to $\mathcal{A}$ are $\mathrm{id}(\alpha)$ and $\mathrm{id}(x\alpha)$; the behavior of $\mathcal{A}'$ when this is not the case is irrelevant. Note that steps 1 and 3 each require at most $2\lceil \log_2 N \rceil - 1$ calls to $\mathcal{B}$ using double-and-add, so $\mathcal{A}'$ makes at most $s - 2$ calls to $\mathcal{B}$.

Let $\gamma_1 = \mathrm{id}(\alpha)$ and $\gamma_2 = \mathrm{id}(x\alpha)$. Without loss of generality we may assume that every interaction between $\mathcal{A}'$ and $\mathcal{B}$ is of the form $\gamma_k = \gamma_i \pm \gamma_j$, with $1 \le i, j < k$, where γ_i and γ_j are group element identifiers that were either inputs or values previously returned by $\mathcal{B}$ (here the notation $\gamma_i \pm \gamma_j$ means that $\mathcal{A}'$ is using $\mathcal{B}$ to add or subtract the group elements identified by γ_i and γ_j). Note that $\mathcal{A}'$ can invert γ_j by computing $\mathrm{id}(0) - \gamma_j$.

The number of such interactions is clearly a lower bound on the number of calls made by $\mathcal{A}'$ to $\mathcal{B}$. To further simplify matters, we will assume that the execution of $\mathcal{A}'$ is padded with operations of the form $\gamma_k = \gamma_1 + \gamma_1$ as required until k reaches s.

Let $N = p^e M$ with $p \perp M$. Define $F_k = a_k X + b_k \in \mathbb{Z}/p^e\mathbb{Z}[X]$ and $z_k \in \mathbb{Z}/M\mathbb{Z}$ via:

$$\begin{array}{lll}
F_1 = 1 & z_1 = 1 & \\
F_2 = X & z_2 = x \bmod M & \\
\vdots & \vdots & \\
F_k = F_i \pm F_j & z_k = z_i \pm z_j & (\text{where } \gamma_k = \gamma_i \pm \gamma_j) \\
\vdots & \vdots & \\
F_s = F_i \pm F_j & z_s = z_i \pm z_j & (\text{where } \gamma_s = \gamma_i \pm \gamma_j)
\end{array}$$

Note that $F_k(x) = \log_\alpha \gamma_k \bmod p^e$ for all k (think of X as the unknown value $x \bmod p^e$).

We now consider the following game, which models the execution of $\mathcal{A}'$. At the start of the game we set $F_1 = 1$, $F_2 = X$, $z_1 = 1$, and set z_2 to a random element of $\mathbb{Z}/M\mathbb{Z}$. We also set γ_1 and γ_2 to distinct random values in $\{0,1\}^m$. For rounds $k = 2, 3, \ldots, m$, the algorithm $\mathcal{A}'$ and the black box $\mathcal{B}$ play the game as follows:

1. $\mathcal{A}'$ chooses a pair of integers i and j, with $1 \le i, j < k$, and a sign $\pm$ that determines $F_k = F_i \pm F_j$ and $z_k = z_i \pm z_j$, and then asks $\mathcal{B}$ for the value of $\gamma_k = \gamma_i \pm \gamma_j$.

2. $\mathcal{B}$ sets $\gamma_k = \gamma_\ell$ if $F_k = F_\ell$ and $z_k = z_\ell$ for some $\ell < k$, and otherwise $\mathcal{B}$ sets γ_k to a random bit-string in $\{0,1\}^m$ that is distinct from γ_ℓ for all $\ell < k$.

After s rounds we pick $t \in \mathbb{Z}/p^e\mathbb{Z}$ at random and say that $\mathcal{A}'$ wins if $F_i(t) = F_j(t)$ for any $F_i \ne F_j$; otherwise $\mathcal{B}$ wins. Notice that the group G also plays no role in the game, it just involves bit-strings, but the constraints on $\mathcal{B}$'s choice of γ_k ensure that the bit strings $\gamma_1, \ldots, \gamma_s$ can be assigned to group elements in a consistent way.

We now claim that

$$\Pr_{x,\mathrm{id},\tau}[\mathcal{A}(\mathrm{id}(\alpha), \mathrm{id}(x\alpha)) = x] \le \Pr_{t,\mathrm{id},\tau}[\mathcal{A}' \text{ wins the game}], \tag{8}$$

where the id function on the right represents an injective map $G \to \{0,1\}^m$ that is compatible with the choices made by $\mathcal{B}$ during the game, i.e. there exists a sequence of group elements $\alpha = \alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_s$ such that $\mathrm{id}(\alpha_i) = \gamma_i$ and $\alpha_k = \alpha_i \pm \alpha_j$, where i, j, and the sign $\pm$ correspond to the values chosen by $\mathcal{A}'$ in the kth round.

For every value of x, id, and τ for which $\mathcal{A}(\mathrm{id}(\alpha), \mathrm{id}(x\alpha)) = x$, there is a value of t, namely $t = x \bmod p^e$, for which $\mathcal{A}'$ wins the game (here we use the fact that $\mathcal{A}'$ always computes $y\alpha$, where $y = \mathcal{A}(\mathrm{id}(\alpha), \mathrm{id}(x\alpha))$, so $\mathcal{A}'$ forces a collision to occur whenever the discrete logarithm is computed correctly, even if $\mathcal{A}$ does not). The number of possible values of $t \in \mathbb{Z}/p^e\mathbb{Z}$ is no greater than the number of possible values of $x \in \mathbb{Z}/N\mathbb{Z}$, hence (8) holds.

We now bound the probability that $\mathcal{A}'$ wins the game. Consider any particular execution of the game, and let $F_{i,j} = F_i - F_j$. We claim that for all i and j such that $F_{i,j} \ne 0$,

$$\Pr_t[F_{i,j}(t) = 0] \le \frac{1}{p}. \tag{9}$$

We have $F_{i,j}(X) = aX + b$ for some $a, b \in \mathbb{Z}/p^e\mathbb{Z}$ with a and b not both zero. Let p^d be the largest power of p that divides both a and b. Let $\overline{a} = a/p^d \bmod p$ and $\overline{b} = b/p^d \bmod p$. Then $\overline{F} = \overline{a}X + \overline{b}$ is not the zero polynomial in $\mathbb{F}_p[X]$. Therefore $\overline{F}$ has at most one root and we have $\Pr[\overline{F}(t \bmod p) = 0] \le 1/p$, which implies $\Pr[F_{i,j}(t) = 0] \le 1/p$, proving (9).

If $\mathcal{A}'$ wins the game then there must exist an $F_{i,j} \ne 0$ for which $F_{i,j}(t) = 0$. Furthermore, since $F_{i,j}(t) = 0$ if and only if $F_{j,i}(t) = 0$, we may assume $i < j$. Thus

$$\begin{aligned}
\Pr_{t,\mathrm{id},\tau}[\mathcal{A}' \text{ wins the game}] &\le \Pr_{t,\mathrm{id},\tau}[F_{i,j}(t) = 0 \text{ for some } F_{i,j} \ne 0 \text{ with } i < j] \\
&\le \sum_{i<j, F_{i,j} \ne 0} \Pr_t[F_{i,j}(t) = 0] \\
&\le \binom{s}{2}\frac{1}{p} < \frac{s^2}{2p},
\end{aligned}$$

where we have used a union bound ($\Pr[A \cup B] \le \Pr(A) + \Pr(B)$) to obtain the sum. □

Corollary 10.6. *Let $G = \langle \alpha \rangle$ be a cyclic group of prime order N. Every deterministic generic algorithm for the discrete logarithm problem in G uses at least $(\sqrt{2} + o(1))\sqrt{N}$ group operations.*

The baby-steps giant-steps algorithm uses $(2 + o(1))\sqrt{N}$ group operations in the worst case, so this lower bound is tight up to a constant factor, but there is a slight gap. In fact, the baby-steps giant-steps method is not quite optimal; the constant factor in the upper bound can be slightly improved via [2] (but this still leaves a small gap).

Let us now extend Theorem 10.5 to the case where the black box also supports the generation of random group elements for a cost of one group operation. We first note that having the algorithm generate random elements itself by computing $z\alpha$ for random $z \in \mathbb{Z}/N\mathbb{Z}$ does not change the lower bound significantly if only a small number of random elements are used; this applies to all of the algorithms we have considered.

Corollary 10.7. *Let $G = \langle \alpha \rangle$ be a cyclic group of prime order N. Every generic Monte Carlo algorithm for the discrete logarithm problem in G that uses $o(\sqrt{N}/\log N)$ random group elements uses at least $(1 + o(1))\sqrt{N}$ group operations.*

This follows immediately from Theorem 10.5, since a Monte Carlo algorithm is required to succeed with probability greater than $1/2$. In the Pollard-ρ algorithm, assuming it behaves like a truly random walk, the number of steps required before the probability of a collision exceeds $1/2$ is $\sqrt{2 \log 2} \approx 1.1774$, so there is again only a small gap in the constant factor between the lower bound and the upper bound.

In the case of a Las Vegas algorithm, we can obtain a lower bound by supposing that the algorithm terminates as soon as it finds a non-trivial collision (in the proof, this corresponds to a nonzero $F_{i,j}$ with $F_{i,j}(t) = 0$). Ignoring the $O(\log N)$ additive term, this occurs within m steps with probability at most $m^2/(2p)$. Summing over m from 1 to $\sqrt{2p}$ and supposing that the algorithm terminates in exactly m steps with probability $(m^2 - (m-1)^2)/(2p)$, the expected number of steps is $2\sqrt{2p}/3 + o(\sqrt{p})$.

Corollary 10.8. *Let $G = \langle \alpha \rangle$ be a cyclic group of prime order N. Every generic Las Vegas algorithm for the discrete logarithm problem in G that generates an expected $o(\sqrt{N}/\log N)$ random group elements uses at least $(2\sqrt{2}/3 + o(1))\sqrt{N}$ expected group operations.*

Here the constant factor $2\sqrt{2}/3 \approx 0.9428$ in the lower bound is once again only slightly smaller than the constant factor $\sqrt{\pi/2} \approx 1.2533$ in the upper bound given by the Pollard-ρ algorithm (under a random walk assumption).

Now let us consider a generic algorithm that generates a large number of random elements, say $R = N^{1/3+\delta}$ for some $\delta > 0$. The cost of computing $z\alpha$ for R random values of z can be bounded by $2R + O(N^{1/3})$. If we let $e = \lceil \lg N/3 \rceil$ and precompute $c\alpha$, $c2^e\alpha$, and $c2^{2e}\alpha$ for $c \in [1, 2^e]$, we can then compute $z\alpha$ for any $z \in [1, N]$ using just 2 group operations. We thus obtain the following corollary, which applies to every generic group algorithm for the discrete logarithm problem.

Corollary 10.9. *Let $G = \langle \alpha \rangle$ be a cyclic group of prime order N. Every generic Las Vegas algorithm for the discrete logarithm problem in G uses an expected $\Omega(\sqrt{N})$ group operations.*

In fact, we can be more precise: the implied constant factor is at least $\sqrt{2}/2$.

References

[1] J.W. Bos, M.E. Kaihara, T. Kleinjung, A.K. Lenstra, and P.L. Montgomery, *PlayStation 3 computing breaks 2^{60} barrier, 112-bit ECDLP solved*, EPFL Laboratory for Cryptologic Algorithms, 2009.

[2] D.J. Bernstein and T. Lange, *Two giants and a grumpy baby*, in Proceedings of the Tenth Algorithmic Number Theory Symposium (ANTS X), Open Book Series **1**, Mathematical Sciences Publishers, 2013, 87–111.

[3] D.E. Knuth, *The Art of Computer Programming, vol. II: Semi-numerical Algorithms*, third edition, Addison-Wesley, 1998.

[4] 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.

[5] V. I. Nechaev, *Complexity of a determinate algorithm for the discrete logarithm*, Mathematical Notes **55** (1994), 165–172.

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

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

[8] V. Shoup, *Lower bounds for discrete logarithms and related problems*, Proceedings of Eurocrypt '97, LNCS **1233** (1997), 256-266, revised version available at `http://www.shoup.net/papers/dlbounds1.pdf`.

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

[10] A.V. Sutherland, *Structure computation and discrete logarithms in finite abelian p-groups*, Mathematics of Computation **80** (2011), 501–538.

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

[12] E. Wenger and P. Wolfger, *Announcement: New 113-bit ECDLP record*, NMBRTHRY listserv, 2015.

18.783 Elliptic Curves

Spring 2015