9 The discrete logarithm problem

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), this means $x\alpha = \beta$. In either case, we call x the discrete logarithm of β with respect to the base α and denote it $\log_\alpha \beta$.[1] 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*:

> *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 $\alpha_1, \dots \alpha_r \in G$ and $n_1, \dots, n_r \in \mathbb{Z}$ 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 $(e_1, \dots, e_r)$ 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* (this means 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 9.1. Suppose $G = \mathbb{F}_{101}^\times$. Then $\log_3 37 = 24$, since $3^{24} \equiv 37 \bmod 101$.

Example 9.2. Suppose $G = \mathbb{F}_{101}^+$. Then $\log_3 37 = 46$, since $46 \cdot 3 \equiv 37 \bmod 101$.

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 9.1 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 9.2 we are working in a group of order 101, which is prime, and in

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

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", its difficulty really depends not on the order of the group (or its structure), but on how the group is explicitly represented. Every group of prime order p is isomorphic to $\mathbb{Z}/p\mathbb{Z}$; computing the discrete logarithm amounts to computing this isomorphism. The reason it is easy to compute discrete logarithms in $\mathbb{Z}/p\mathbb{Z}$ has nothing to do with the structure of $\mathbb{Z}/p\mathbb{Z}$ as an additive group, rather it is the fact that $\mathbb{Z}/p\mathbb{Z}$ also use a ring structure; in particular, it is a Euclidean domain, which allows us to use the extended Euclidean algorithm to compute multiplicative inverses. This involves operations (multiplication) other than the standard group operation (addition), 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 group. 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.

9.1 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 an abstract group G solely through a *black box* (sometimes called an *oracle*). All group elements are opaquely encoded as bit-strings via a map $\mathsf{id}\colon G \to \{0,1\}^m$ chosen by the black box. The black box supports the following operations.

1. `identity`: output $\mathsf{id}(1_G)$.
2. `inverse`: given input $\mathsf{id}(\alpha)$, output $\mathsf{id}(\alpha^{-1})$.
3. `composition`: given inputs $\mathsf{id}(\alpha)$ and $\mathsf{id}(\beta)$, output $\mathsf{id}(\alpha\beta)$.
4. `random`: output $\mathsf{id}(\alpha)$ for a uniformly distributed random element $\alpha \in G$.

In the description above we used multiplicative notation; in additive notation we would have outputs $\mathsf{id}(0_G)$, $\mathsf{id}(-\alpha)$, $\mathsf{id}(\alpha-\beta)$ for the operations `identity`, `inverse`, `composition`, respectively.

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 the identification map $\mathrm{id}\colon G \to \{0,1\}^m$ used by the black box is injective. With uniquely identified group elements we can test equality by simply comparing identifiers, without needing to consult the black box.[2]

The black box is allowed to use *any* injective identification map (e.g., a random one). A *generic algorithm* cannot depend on a particular choice of the identification map; this prevents it from taking advantage of how group elements are represented. We have already seen several examples of generic group algorithms, including various exponentiation algorithms, fast order algorithms, and the baby-steps giant-steps method.

[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.

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 so "for free". The implicit assumption is that the cost of any useful auxiliary computations are at worst proportional to the number of group operations — this is true of all the algorithms we will consider.

9.2 Generic algorithms for the discrete logarithm problem

We now consider generic algorithms for the discrete logarithm problem in the standard setting of a cyclic group $\langle\alpha\rangle$. We shall assume throughout that $N := |\alpha|$ is known. This is a reasonable assumption for three reasons: (1) in cryptographic applications it is quite important to know N (or at least to know that N is prime), (2) the lower bound we shall prove applies even when the algorithm is given N, (3) for a generic algorithm, computing $|\alpha|$ is strictly easier than solving the discrete logarithm problem [12], and in most cases of practical interest (the group of rational points on an elliptic curve over a finite field, for example), there are (non-generic) polynomial time algorithms to compute N.

The cyclic group $\langle\alpha\rangle$ is isomorphic to the additive group $\mathbb{Z}/N\mathbb{Z}$. For 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 inside 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):

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

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 $x = \log_\alpha \beta$. In order to simplify our notation we will write the group operation additively, so that $\beta = x\alpha$.

9.3 Linear search

Starting from α, compute

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

and then output x (or if we reach $N\alpha$ with out finding β, 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 $O(1)$ group elements.

9.4 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 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 we 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, if we assume that $\beta \in \langle\alpha\rangle$ then we do not even need to know N: this is clear for the linear search, and for the baby-steps giant-steps method we could simply start by assuming $N = 2$ and if/when that fails, keep doubling N and rerunning the algorithm until we succeed. This still yields an $O(\sqrt{N})$ complexity.[3]

For the next algorithm we consider it is quite important to know N exactly, in fact we will assume that we know its prime factorization; factoring N does not require any group operations, so in our complexity model which counts group operations, a generic algorithm can factor any integer N "for free". In practical terms, there are algorithms to factor N that are much faster than the generic lower bound we will prove below; as we will see in the next lecture, the elliptic curve factorization method is one such algorithm.

9.5 The Pohlig-Hellman algorithm

We now introduce the Pohlig-Hellman[4] 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.

[3] There are more efficient ways to do an unbounded baby-steps giant-steps search, see [12, 14].

[4] The article by Pohlig and Hellman [6] notes that essentially equivalent versions of the algorithm were independently found by R. Silver, and by R. Schroeppel and H. Block, none of whom published the result.

We first reduce to the prime power case. Suppose $N = N_1N_2$ with $N_1 \perp N_2$. 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 via

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

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; with the fast Euclidean algorithm the time to compute M_1 and M_2 is $O(\mathsf{M}(n)\log n)$ bit operations, where $n = \log N$.

Let us now consider the computation of $x = \log_\alpha \beta$. Define

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

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 the scalar multiplication 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 (since $N_1 \perp N_2$). From (1) and (2) we have $M_1 \equiv 1 \bmod N_1$ and $M_2 \equiv 0 \bmod N_1$, so the second term in (3) vanishes and the first term can be simplified, yielding

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

We similarly find that $N_1\beta = x_2N_1\alpha$. Therefore

$$\begin{aligned} x_1 &= \log_{N_2\alpha} N_2\beta, \\ x_2 &= \log_{N_1\alpha} N_1\beta. \end{aligned}$$

If we know x_1 and x_2 then we can compute $x = (M_1x_1 + M_2x_2) \bmod N$. Thus the computation of $x = \log_\alpha \beta$ can be reduced to the computation of $x_1 = \log_{N_2\alpha} N_2\beta$ and $x_2 = \log_{N_1\alpha} N_1\beta$. If N is a prime power this doesn't help (either $N_1 = N$ or $N_2 = N$), but otherwise these two discrete logarithms involve groups of smaller order. In the best case $N_1 \approx N_2$, and we reduce our original problem to two subproblems of half the size, and this reduction involves only $O(n)$ groups operations (the time to compute $N_1\alpha, N_1\beta, N_2\alpha, N_2\beta$ using double-and-add scalar multiplication).

By applying the reduction above recursively, we can reduce 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

$$\begin{aligned} \beta &= (x_0 + p^{e_0}x_1)\alpha, \\ p^{e_1}\beta &= x_0p^{e_1}\alpha + x_1p^e\alpha. \end{aligned}$$

The second term in the last equation is zero, since α has order $N = p^e$, so

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

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

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

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) using $O(n)$ group operations.

The Pohlig-Hellman method [6] recursively applies the two reductions above to reduce the problem to a set of discrete logarithm computations in groups of prime order.[5] 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.

9.6 Complexity analysis

Let $N = p_1^{e_1} \cdots p_r^{e_r}$ be the prime factorization of N. 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 won't use this). The exponents e_i are all bounded by $\lg N = O(n)$, thus reducing prime powers to the prime case involves at most an additional $O(\log n)$ levels of recursion, since the exponents are reduced by roughly a factor of 2 at each level.

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 prime powers $p_i^{e_i}$ vary widely in size, this bound still holds.

The product of the orders of the bases used at any given layer of the recursion tree never exceeds N. The number of group operations required at each internal node of the recursion tree is linear in the bit-size of the order of the base, since only $O(1)$ scalar multiplications are used 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. If we use the baby-steps giant-steps algorithm to handle the prime order cases, each leaf uses $O(\sqrt{p_i})$ group operations and 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 standard Pohlig-Hellman approach is deterministic.

[5] 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 [9, §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 [13]).

The Pohlig-Hellman algorithm can be extremely efficient when N is composite; if N is sufficiently smooth its running time is quasi-linear in $n = \log N$, comparable to the cost of 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.

9.7 Randomized algorithms for the discrete logarithm problem

So far we have only considered deterministic algorithms for the discrete logarithm problem. We now consider a probabilistic approach. 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).

9.7.1 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 by increasing the number of balls by a constant factor. 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 corresponding to linear combinations of α and β into N bins corresponding to the elements 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 computed $\sqrt{2N}$ random linear combinations of α and β, 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 would need to keep a record of all the linear combinations we have computed, which takes 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.

9.7.2 Random walks on a graph

We now want to view the group $G = \langle\alpha\rangle$ as the vertex set V of a connected graph Γ whose edges $e_{ij} = (\gamma_i, \gamma_j)$ are labeled with the group element $\delta_{ij} = \gamma_j - \gamma_i$ satisfying $\gamma_i + \delta_{ij} = \gamma_j$ (a Cayley graph, for example). If we know how to express each δ_{ij} as a linear combination of α and $\beta \in \langle\alpha\rangle$, then any cycle in Γ yields a linear relation involving α and β. Provided the coefficient of β is invertible modulo $N := |\alpha|$, we can use this relation to compute $\log_\alpha \beta$.

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 $i \geq 0$. Note that V is finite, so this must happen eventually.

Our random walk 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 $v_\lambda, v_{\lambda+1}, \ldots, v_{\rho-1}$. This 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.

In order to extract information from this cycle we need to augment the function f so that we can associate linear combinations $a\alpha + b\beta$ to each edge in the cycle. But let us first compute the expected number of steps a random walk takes to reach its first collision.

Theorem 9.3. *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

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

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

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

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 $\log(1-x)$.

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

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

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

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

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

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

Remark 9.4. 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.

9.8 Pollard-ρ Algorithm

We now present the Pollard-ρ algorithm for computing $\log_\alpha \beta$, given $\beta \in \langle\alpha\rangle$; we should note that the assumption $\beta \in \langle\alpha\rangle$ which was not necessary in the baby-steps giant-steps algorithm is crucial here. As noted earlier, finding 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. We thus extend the function $f\colon G \to G$ used to define 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 we require to have the property that if the input (a, b, γ) satisfies $a\alpha + b\beta = \gamma$, then $(a', b', \gamma') = f(a, b, \gamma)$ should satisfy $a'\alpha + b'\beta = \gamma'$.

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 $r \approx 20$ works well [15]. We 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*

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

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 take the integer whose base-2 representation corresponds to the identifier $\mathsf{id}(\gamma) \in \{0,1\}^m$ and reduce it modulo r.[6]

[6]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.

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 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 the function f does define a walk in the same sense as before. We now give the algorithm.

Algorithm 9.5 (Pollard-ρ)**.** Given α, $N = |\alpha|$, $\beta \in \langle\alpha\rangle$, compute $\log_\alpha \beta$ as follows:

1. Compute $\delta_i = c_i\alpha + d_i\beta$ for $r \approx 20$ randomly chosen pairs $c_i, d_i \in \mathbb{Z}/N\mathbb{Z}$.
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 $\gamma_k = \gamma_j$ with $k > j$.
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 start over at step 1.

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, even though it is often referred to in the literature as a Monte Carlo algorithm, due to the title of [8].

The description above does not specify 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. It is thus 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.

9.9 Floyd's cycle detection method

Floyd's cycle detection method [5, Ex. 3.1.6, p. 7] 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 ρ-shaped walk. They start with the same γ_0, but in each iteration the hare takes two steps, while the tortoise takes just one. We thus modify step 3 of Algorithm 9.5 to 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 (which must already be in the cycle) is guaranteed to collide with the tortoise within σ iterations, where σ is the length of the cycle (to see this, note that the hare gains on the tortoise by one step in each iteration and cannot pass the tortoise without landing on it). On average, we expect it to take $\sigma/2$ 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 $\mathrm{E}[\lambda + \sigma/2] = 3/4\,\mathrm{E}[\rho]$. But each iteration now requires three group operations, so the algorithm is actually slower by a factor of 9/4. Still, this achieves a time complexity of $O(\sqrt{N})$ group operations while storing just $O(1)$ group elements, which is a dramatic improvement.

9.10 The method of distinguished points

The "distinguished points" method (commonly attributed to Ron 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), and uses just one group operation for each iteration, rather then the three required by Floyd's method.

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 will store all the distinguished we encounter during our walk, but we want our walk to contain many distinguished points; say $(\log N)^c$, on average, for some constant $c > 0$. This means we should choose B so that

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

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. 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 9.6 (Pollard-ρ using distinguished points)**.**

1. Pick random $c_i, d_i, a_0, b_0 \in \mathbb{Z}/N\mathbb{Z}$, compute $\delta_i = c_i\alpha + d_i\beta$ and $\gamma_0 = a_0\alpha + b_0\beta$, and initialize $D \leftarrow \emptyset$.
2. 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 at step 1 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 massively parallel implementation, which is critical for any large-scale discrete logarithm computation. 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 processors 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. However, the individual processors cannot realize this themselves, since they only know the distinguished points they have seen, not those seen by other processors. 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 9.6 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.

9.11 Current ECDLP records

The current record for computing discrete logarithms on elliptic curves over finite fields involves a cyclic group with 117-bit prime order on an elliptic curve $E/\mathbb{F}_q$ with $q = 2^{127}$ and was set in 2016. The computation was run on 576 XC6SLX150 FPGAs and took about 200 days [1]. The current record for elliptic curves over prime fields was set in 2017 using the curve $E : y^2 = x^3 + 3$ over the 114-bit prime field $\mathbb{F}_{11957518425389075254535992784167879}$ with $\#E(\mathbb{F}_p)$ prime. This computation took advantage of the extra automorphisms of this curve and took the equivalent of 81 days running on 2000 Intel cores [4]. The record for elliptic curves over prime fields without extra automorphisms was set in 2009 using a 112-bit prime order group on an elliptic curve $E/\mathbb{F}_p$ with $p = (2^{128} - 3)/(11 \cdot 6949)$; this computation was run on a cluster of 200 PlayStation 3 consoles and took 180 days [3]. All of these records were set using a parallel Pollard-rho search and the method of distinguished points.

We should note that for elliptic curves over non-prime fields the non-generic methods we will discuss in the next lecture (index calculus) can be applied. This changes the situation dramatically, and it is now practical to solve the discrete logarithm problem on an elliptic curves over $\mathbb{F}_q$ for suitably composite q with thousands of bits. But for elliptic curves over prime fields we know of no methods other than generic algorithms.

This claim holds even for quantum computers: there are very efficient algorithms for solving the discrete logarithm problem on an elliptic curve over a prime field, but these algorithms are generic in the sense that they apply to any group for which the group operation can be effectively implemented on a quantum computer using unique representations of group elements, an assumption that is already implicit in our black box model.

9.12 Computing discrete logarithms via the hidden subgroup problem

While we won't discuss quantum computing in this course (take 18.435J), let us briefly describe an efficient generic algorithm for solving the discrete logarithm on a quantum computer. As first proposed by Peter Shor [10] for computing discrete logarithms in $\mathbb{F}_p^\times$ and then generalized by others, this involves a reduction to what is now known as the *hidden subgroup problem* (HSP). We are given a finite group G containing a subgroup H along with a function $f\colon G \to S$ that is constant on cosets of H and maps each coset to a distinct element of S; here S can be any finite set, but for us S will actually be the group we want to compute a discrete logarithm in.

The hidden subgroup problem is to compute a set of generators for the unknown group H using f and group operation in G. There is an efficient polynomial-time algorithm to solve this problem on a quantum compute when H is abelian[7] assuming the group operation in G can be efficiently implemented on a quantum computer.[8] We won't describe the quantum algorithm for solving the hidden subgroup problem here, our aim is simply to show how it can be used to easily solve the discrete logarithm problem.

[7] The hidden subgroup problem for non-abelian groups is still open; even for dihedral groups we do not have a quantum polynomial-time algorithm.

[8] One can encapsulate this assumption by postulating a "quantum black box" that is used by "quantum generic group algorithms", just as we did for classical generic group algorithms above.

To compute the discrete logarithm problem of $\beta = \alpha^x$ in the cyclic group $\langle\alpha\rangle$ of order N one defines G, H, S, and f as follows:

$$G \coloneqq \mathbb{Z}/N\mathbb{Z} \times \mathbb{Z}/N\mathbb{Z}, \qquad H \coloneqq \langle(x,1)\rangle, \qquad S \coloneqq \langle\alpha\rangle, \qquad \begin{aligned} f &\colon G \to S \\ (a,b) &\mapsto b\beta - a\alpha \end{aligned}$$

The computation of f only requires the inputs α, β and operations in the group $\langle\alpha\rangle$, it does not require knowledge of H or the discrete logarithm x we are tying to compute. We can use the standard double-and-add algorithm to compute f using $O(n)$ group operations. Given any set of generators for H we can easily recover x. All we need is an element $(a,b) \in H$ with $b \perp N$, since $x = ab^{-1} \bmod N$; if N is prime any nonzero element of H will do, and in general we can easily construct such an element as a linear combination of whatever set of generators our quantum computer gives us.

9.13 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 bounded above $1/2$, and the expected running time of any generic Las Vegas algorithm is $\Omega(\sqrt{p})$ group operations.

The following theorem due to Shoup [11] generalizes an earlier result of Nechaev [7]. 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 map $\mathsf{id}\colon G \hookrightarrow \{0,1\}^m$, where $m = O(\log|G|)$.

Theorem 9.7 (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 $\mathsf{id}\colon G \hookrightarrow \{0,1\}^m$. 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 $s - 4\lceil\lg N\rceil$ calls to $\mathcal{B}$, for some integer s, and let x denote a random element of $\mathbb{Z}/N\mathbb{Z}$. Then*

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

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 $z \in \mathbb{Z}/N\mathbb{Z}$ (using at most $2\lg N$ group operations). We assume that $\mathcal{A}$ is given the group order N (this only makes the theorem stronger). The theorem includes deterministic algorithms as a special case where $\mathcal{A}$ does not use any of the random bits in τ. Bounding the number of calls $\mathcal{A}$ makes to $\mathcal{B}$ might appear to make the theorem inapplicable to Las Vegas algorithms, but we can convert a Las Vegas algorithm to a Monte Carlo algorithm by forcing it to halt and generate a random output if it exceeds its expected running time by some constant factor.

In order to simplify the presentation we will only prove Theorem 9.7 in the case $N = p$ is prime; the proof for composite N is an easy generalization of the prime order case, which in some sense the only case that matters given our $O(n \log n + n\sqrt{p})$ upper bound ($n = \log N$).

Proof of Theorem 9.7 for $N = p$ prime. To simplify the proof, we will replace $\mathcal{A}$ by an algorithm $\mathcal{A}'$ that does the following:

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

In the description above we assume that the inputs to $\mathcal{A}$ are $\mathsf{id}(\alpha)$ and $\mathsf{id}(x\alpha)$; 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 $\gamma_1 = \mathsf{id}(\alpha)$ and $\gamma_2 = \mathsf{id}(x\alpha)$. 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 identifiers of group elements that are 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 $\mathsf{id}(0) - \gamma_j$.

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.

For $k = 1, \dots, s$ define $F_k = a_k X + b_k \in \mathbb{Z}/p\mathbb{Z}[X]$ via:

$$F_1 := 1, \qquad F_2 := X, \qquad F_k := F_i \pm F_j \qquad (2 < k \le s).$$

Each F_k is a linear polynomial in X that satisfies

$$F_k(x) \equiv \log_{\gamma_1} \gamma_k \bmod p,$$

where we are abusing notation by writing $\gamma_k = \mathsf{id}(g_k)$ in place of $g_k \in G$.

Let us 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 $k = 2, 3, \dots, s$, 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 then asks $\mathcal{B}$ for the value of $\gamma_k = \gamma_i \pm \gamma_j$.
2. $\mathcal{B}$ sets $\gamma_k = \gamma_{k'}$ if $F_k = F_{k'}$ for some $k' < k$, and otherwise $\mathcal{B}$ sets γ_k to a random bit-string in $\{0,1\}^m$ that is distinct from $\gamma_{k'}$ for all $k' < k$.

After the sth round we pick $t \in \mathbb{Z}/p\mathbb{Z}$ 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 $\gamma_1, \dots, \gamma_s$ can be assigned to group elements in a consistent way. We now claim that

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

where the id function on the right represents an injective map $G \hookrightarrow \{0,1\}^m$ that is compatible with the choices made by $\mathcal{B}$ during the game, in other words, there exists a sequence of

group elements $\alpha = \alpha_1, \alpha_2, \alpha_3, \ldots, \alpha_s$ such that $\mathsf{id}(\alpha_i) = \gamma_i$ 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.

Any triple (x, id, τ) for which $\mathcal{A}(\mathsf{id}(\alpha), \mathsf{id}(x\alpha)) = x$ is also a triple (t, id, τ) for which $\mathcal{A}'$ wins the game; here we use the fact that $\mathcal{A}'$ always computes $y\alpha$, where $y = \mathcal{A}(\mathsf{id}(\alpha), \mathsf{id}(x\alpha))$, so $\mathcal{A}'$ forces a collision to occur whenever $\mathcal{A}$ outputs the correct value of x even if $\mathcal{A}$ did not actually encounter a collision (maybe $\mathcal{A}$ just made a lucky guess). Thus (8) holds.

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

$$\Pr_t[F_{ij}(t) = 0] \leq \frac{1}{p}. \tag{9}$$

We have $F_{ij}(X) = aX + b$ for some $a, b \in \mathbb{Z}/p\mathbb{Z}$ with a and b not both zero. If a is zero then $F_{ij}(t) = b \neq 0$ for all $t \in \mathbb{Z}/p\mathbb{Z}$ and (9) holds. Otherwise the map $[a]\colon t \mapsto at$ is a bijection, and in either case there is at most one value of t for which $F_{ij}(t) = 0$, which proves (9).

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

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

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

Corollary 9.8. *Let G 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 2 in the upper bound $(2 + o(1))\sqrt{N}$ can be improved via [2] (but this still leaves a small gap).

Let us now extend Theorem 9.7 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 9.9. *Let G 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 9.7, since a Monte Carlo algorithm is required to succeed with probability bounded above $1/2$. 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_{ij} with $F_{ij}(t) = 0$). 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 9.10. *Let G 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.*

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 9.11. *Let G 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] Daniel J. Bernstein, Susannne Engles, Tanja Lange, Ruben Niederhagen, Christof Paar, Peter Schwabe, and Ralf Zimmerman, *Faster elliptic curve discrete logarithms on FPGAs*, Cryptology eprint Archive, Report 2016/382, 2016.

[2] Daniel J. Bernstein and Tanya 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] Joppe W. Bos, Marcelo E. Kaihara, Thorsten Kleinjung, Arjen K. Lenstra, and Peter L. Montgomery, *PlayStation 3 computing breaks 2^{60} barrier, 112-bit ECDLP solved*, EPFL Laboratory for Cryptologic Algorithms, 2009.

[4] Takuya. Kusaka, Sho Joichi, Ken Ikuta, Md. Aal-Amin Khandaker, Yasuyuki Nogami, Satoshi Uehara, Nariyoshi Yamai, *Solving 114-bit ECDLP for a Barreto–Naehrig curve*, Information Security and Cryptology – ICISC 2017, LNCS **10779** (2018), 231–244.

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

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

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

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

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

[10] Peter Shor, *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer*, SIAM J. Computing **26** (1997), 1484–1509.

[11] Victor 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`.

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

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

[14] David C. Terr, *A modification of Shanks baby-step giant-step method*, Math. Comp. **69** (2000), 767–773.

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