Lecture 7 – More algorithms for the Discrete Logarithm Problem

Chloe Martindale

2023

These notes are for the lecture course Cryptology at the University of Bristol. If you come across any typos, please email chloe.martindale@bristol.ac.uk.

7.1 Baby-Step-Giant-Step

We have already seen that if you want to find discrete logarithms in $\mathbb{F}_p^*$ and $p-1$, the size of the multiplicative group $\mathbb{F}_p^*$, has only small factors, you can do this very effectively using Pohlig-Hellman.

However, if you choose a finite field $\mathbb{F}_p$ such that there is a large prime ℓ dividing $p-1$, then you can also find an element $g \in \mathbb{F}_p$ of order ℓ where Pohlig-Hellman won't help you. So the question is: can we do better than brute-force? Below we will see two algorithms, Baby-Step-Giant-Step and Pollard-ρ, that are essentially clever methods for brute-forcing.

As always, we want to break the discrete logarithm problem, so suppose you have $g \in \mathbb{F}_p^*$ of order ℓ (not necessarily prime). Then, given g and g^a, find a. Remember that changing a by adding a multiple of ℓ amounts to multiplying g^a by $g^\ell = 1$, so it suffices to compute $a \pmod{\ell}$.

The algorithm is as follows:

1. For i from 0 to $\sqrt{\ell}$, compute and save $b_i = g^i$.
2. For j from 0 to $\sqrt{\ell}+1$, compute $c_j = g^a \cdot g^{-\sqrt{\ell} \cdot j}$; break if there exists an i such that $c_j = b_i$.
3. Return $a = i + \sqrt{\ell} \cdot j$.

Example Compute a such that $3^a \equiv 37 \pmod{101}$. The order of 3 in $\mathbb{F}_{101}^*$ is 100, so $\ell = 100$ and $\sqrt{\ell} = 10$. From step 1, the 'baby step', we get a table of values

i	0	1	2	3	4	5	6	7	8	9	10
$b_i = 3^i \pmod{101}$	1	3	9	27	81	41	22	66	97	89	65

Note that computing this table costs $10 = \sqrt{\ell}$ multiplications.

For step 2, the 'giant step', $c_0 = g^a$ is easy. For $c_1 = 3^a \cdot 3^{-10}$, we have to compute one inversion and exponentiate to get the 10th power, but we save the values c_1 and 3^{-10}. For

$c_2 = 3^a \cdot 3^{-20}$, we first observe that $c_2 = c_1 \cdot 3^{-10}$, and we saved the values c_1 and 3^{-10}, so this just costs one multiplication, as will the computation of every c_j after this by a similar argument. So we get

$$\begin{aligned} c_0 &= 37, \\ c_1 &= 13, \\ c_2 &= 81, \end{aligned}$$

at which point we stop because $c_2 = b_4$, or in order words

$$3^a \cdot 3^{-20} \equiv 3^4 \pmod{101},$$

so $a = 24$.

As is illustrated in the example, the baby-step-giant-step algorithm costs at most $2\sqrt{\ell}$ multiplications plus some set up costs for the inversion and exponentiation to the $\sqrt{\ell}^{\text{th}}$ power (for example using square-and-multiply. As ℓ grows, these setup costs become negligable, so we say that 'the complexity of baby-step-giant-step is about $O(\sqrt{\ell})$' – any constants multiplying the $\sqrt{\ell}$ or added to it disappear in the big O.

This gives a square root speed up on just brute forcing – if your ℓ has 2000 bits then $\sqrt{\ell}$ only has 1000 bits – but at the expense of a pretty serious memory assumption: baby-step-giant-step also requires $O(\sqrt{\ell})$ storage. The next algorithm solves that problem.

7.2 Pollard's ρ method

Again, we want to solve the discrete logarithm problem: given g and $g^a \in \mathbb{F}_p^*$, find a.

The aim of Pollard's ρ method is to output integers $b, c, b', c' \in \{1, \ldots, \ell\}$ such that $c \neq c'$ and

$$g^b (g^a)^c = g^{b'} (g^a)^{c'}. \tag{7.1}$$

Why does this solve our problem? Well, suppose that the order of g in $\mathbb{F}_p^*$ is ℓ. Then, as before, adding ℓ to the exponent is equivalent to multiplying by $g^\ell = 1$, so taking logarithms of our equation (7.1) gives us:

$$b + ac \equiv b' + ac' \pmod{\ell}.$$

Rearranging this equation then gives us the secret a:

$$a \equiv \frac{b - b'}{c' - c} \pmod{\ell},$$

thus solving the discrete logarithm problem.

So, how exactly do we find such b, c, b', and c'? To do this, we define a *graph* G with vertices $G_i \in \mathbb{F}_p$ such that for each i there exists b_i and c_i such that $G_i = g^{b_i} g^{ac_i}$. We define G_i, b_i, and c_i iteratively, and once we've found $i \neq j$ with $G_i = G_j$, we have candidates for b, c, b', c' satisfying (7.1), namely $b = b_i$, $c = c_i$, $b' = b_j$, $c' = c_j$.

The iterative sequence that turns out to be the most efficient to do this is:

$$G_0 = g,\ b_0 = 1,\ c_0 = 0,$$

and

$$(G_{i+1}, b_{i+1}, c_{i+1}) = \begin{cases} (G_i \cdot g, b_i + 1, c_i) & G_i \equiv 0 \pmod 3, \\ (G_i \cdot g^a, b_i, c_i + 1) & G_i \equiv 1 \pmod 3, \\ (G_i^2, 2b_i, 2c_i) & G_i \equiv 2 \pmod 3. \end{cases}$$

You might think the $\pmod 3$ looks a bit random, but this is basically just a way of making sure the choice of how to iterate changes around a bit.

Example. Compute a such that $3^a \equiv 7 \pmod{17}$. The order of 3 in $\mathbb{F}_{17}$ is 16, so $\ell = 16$. The algorithm above outputs a list:

$$\begin{aligned} G_0, b_0, c_0 &= 3, 1, 0 \\ G_1, b_1, c_1 &= 9, 2, 0 \\ G_2, b_2, c_2 &= 10, 3, 0 \\ G_3, b_3, c_3 &= 2, 3, 1 \\ G_4, b_4, c_4 &= 4, 6, 2 \\ G_5, b_5, c_5 &= 11, 6, 3 \\ G_6, b_6, c_6 &= 2, 12, 6, \end{aligned}$$

at which point we terminate because $G_6 = G_3$. In particular, this means that

$$g^{b_6} \cdot (g^a)^{c_6} = g^{b_3} \cdot (g^a)^{c_3},$$

which plugging in the values gives

$$3^3 \cdot (3^a) \equiv 3^{12} \cdot (3^a)^6,$$

giving $a \equiv 9 \cdot (-5)^{-1} \equiv 11 \pmod{16}$.

Pollard's ρ algorithm terminates after about $\sqrt{\frac{\pi}{2}\ell}$ steps, so also costs $O(\sqrt{\ell})$ multiplications, but requires only constant storage, which we typically notate by $O(1)$, so is better than baby-step-giant-step for memory reasons and is the algorithm typically used in practise.

Both baby-step-giant-step and Pollard ρ are what we refer to as 'generic' algorithms: they're not using anything particular about the structure of the group $\mathbb{F}_q^*$ or the choice of g for example–both are essentially just same methods for brute forcing.

In our context, that is in finite fields, there is another algorithm that beats both of these generic algorithms.

7.3 Index calculus

Again, we want to solve the discrete logarithm problem: given g and $g^a \in \mathbb{F}_q$, find $a = \log_g(g^a)$. This is the last algorithm we will see to attack this problem (and also the most efficient in this setting). We will first look at an example and then work out how to write

down a general algorithm from that example.

Example Suppose you are given that 17 has order 106 in $\mathbb{F}_{107}^*$, and that $17^a \equiv 91 \pmod{107}$, and you want to compute a, i.e., you want to compute $\log_{17}(91)$. With the index calculus algorithm, the first thing you do is choose a *factor base* $\mathcal{F}$, which can contain any (and as many) primes (as) you like; here we will choose

$$\mathcal{F} = \{2, 3, 5\}.$$

We then compute $\log_{17}(n)$ for every $n \in \mathcal{F}$ in the following way: compute and factorise $17^i \pmod{107}$ for increasing i until you have found $3 = |\mathcal{F}|$ equations for the $\log_{17}(n)$. In this example, we get

$$17^2 \equiv 3 \cdot 5^2 \pmod{107},$$

which taking logs gives

$$2 \equiv \log_{17}(3) + 2\log_{17}(5) \pmod{106}, \tag{7.2}$$

then $17^3, \ldots, 17^8$ all have factors which are not in $\mathcal{F}$, but

$$17^9 \equiv 2^2 \cdot 5 \pmod{107},$$

which taking logs gives

$$9 \equiv 2\log_{17}(2) + \log_{17}(5) \pmod{106}, \tag{7.3}$$

and finally

$$17^{11} \equiv 2 \pmod{107},$$

which taking logs gives

$$11 \equiv \log_{17}(2) \pmod{106}. \tag{7.4}$$

Solving the three simultaneous equations (7.2), (7.3), and (7.4) gives

$$\begin{aligned} \log_{17}(2) &\equiv 11 \pmod{106}, \\ \log_{17}(3) &\equiv 28 \pmod{106}, \\ \log_{17}(5) &\equiv 93 \pmod{106}. \end{aligned}$$

So now we've found these values, what do we do with them? We want to be able to write the discrete log we're actually interested in, namely $\log_{17}(91)$, in terms of the discrete logs we now know, namely $\log_{17}(n)$ for $n \in \mathcal{F}$, and of course $\log_{17}(17^j)$ $(= j)$ for small values of j. We can play the same game as above: try multiplying 91 with 17^j for small values of j and factorizing until we find a number with only factors from our factor base. Doing this we see that $17^0 \cdot 91, \ldots, 17^4 \cdot 91$ yields nothing but

$$17^5 \cdot 91 \equiv 2^2 \cdot 5^2 \pmod{107},$$

which taking logs gives

$$5 + \log_{17}(91) \equiv 2\log_{17}(2) + 2\log_{17}(5) \pmod{106},$$

and plugging in the values above this gives us that

$$a = \log_{17}(91) = 97.$$

So, let's summarize our method into a more general algorithm. Suppose you are given g and $g^a \in \mathbb{F}_p$ and you want to compute a. Then

1. Choose your factor base $\mathcal{F} = \{p_1, \ldots, p_n\}$.
2. Compute, for each $i = 1, \ldots, n$, $\log_g(p_i)$:
   (a) For increasing $j \geq 1$, factorize g^j. Break when you have found n values of j for which all the factors of g^j are in $\mathcal{F}$.
   (b) Take logs of the n equations for values g^j with all factors in $\mathcal{F}$ to get n simultaneous equations for $\log_g(p_1), \ldots, \log_g(p_n)$.
   (c) Solve your n simultaneous equations to get $\log_g(p_1), \ldots, \log_g(p_n)$.
3. For increasing $j \geq 0$, factoring $g^j \cdot g^a$. Break when all factors of j are in $\mathcal{F}$.
4. Take logs of the equation from the previous step, and solve for a.

This algorithm is by far the most efficient known algorithm for this setting of the discrete logarithm problem. To write down the complexity, we recall the notation:

$$L_N(\alpha, c) = e^{c \log N^\alpha \log\log N^{1-\alpha}},$$

where $\alpha \in [0, 1]$ Recall also that the closer α is to 0, the closer an algorithm is to being polynomial time, and the closer it is to 1, the closer an algorithm is to being exponential time.

The most optimized version of the index calculus algorithm (containing many many details not covered here) has complexity $L_p(1/3, c)$, where the constant c depends very heavily on the conditions, so that's closer to the polynomial time end than the exponential end, but the different is still (asymptotically) big enough for powers of large primes that it is possible to make use of finite fields in cryptography by scaling up the numbers. In particular, scaling up p to at least 3000 bits for 128-bit security, meaning that it should take about 2^{128} bit operations to break the protocol. Compare this to our Pollard ρ algorithm which takes about $\sqrt{p}$ bit operations to break the protocol–so we would need p to be about 256 bits, and you see how much different the index calculus makes.

There are other examples of groups in which the index calculus is less or not at all effective. These groups, namely elliptic curve groups, are currently the most commonly used in practise–the size of your group can indeed be only about 256 bits instead of 3000. But that is beyond the scope of this course!