Discrete logarithm

Number theory

Application of discrete models

1 Prime modulus

From the previous part (Modular arithmetic), we know that if the modulus (m) is a prime number, then certain operations are simpler, e.g. we can divide by all nonzero elements.

Let's examine $\mathbb{Z}_p$ further, where p is a prime number. In particular, consider its nonzero elements:

Definition 1.1. For a prime number p, let $\mathbb{Z}_p^*$ be those elements of $\mathbb{Z}_p$ which do not contain zero, i.e. $\mathbb{Z}_p^* := \mathbb{Z}_p \setminus \{\overline{0}\}$.

For example, $\mathbb{Z}_5^* = \{\overline{1}, \overline{2}, \overline{3}, \overline{4}\}$, and in general, $\mathbb{Z}_p^* = \{\overline{1}, \overline{2}, \ldots, \overline{p-1}\}$ (so $\mathbb{Z}_p^*$ has exactly p–1 elements).

Why is this interesting?

Theorem 1.2. *If p is prime, then $\forall \overline{a}, \overline{b} \in \mathbb{Z}_p^* : \overline{a} \cdot \overline{b} \in \mathbb{Z}_p^* \ \wedge \ \overline{a}/\overline{b} \in \mathbb{Z}_p^*$.*

This means that we can always multiply and the divide within $\mathbb{Z}_p^*$, and the result will also be in this set. For multiplication, it means that the product will never be zero. But this is only true for prime modulus! For example, if the modulus is $m = 10$, then $\overline{4} \cdot \overline{5} = \overline{20} = \overline{0}$. (Later we'll call such numbers *zero divisors*.) Also, this is only true for multiplication and divisor, not for addition and subtraction, e.g. if $p = 5$, we have $\overline{3} + \overline{2} = \overline{0}$.

2 Modular exponentiation

In the previous part, we discussed the basics of modular arithmetic: modular addition, subtraction, multiplication and division. Now let's see modular exponentiation.

Exponentiation for a positive integer exponent is just repeated multiplication, e.g. for integers, $5^3 = 5 \cdot 5 \cdot 5 = 125$. Modular exponentation is the same, e.g. in $\mathbb{Z}_7$: $\overline{5}^3 = \overline{5} \cdot \overline{5} \cdot \overline{5} = \overline{4} \cdot \overline{5} = \overline{6}$.

For integer exponents (i.e. positive, zero or negative integers), it can be generalized in the usual way:

Definition 2.1. Let $m \in \mathbb{N}^+, \overline{a} \in \mathbb{Z}_m, k \in \mathbb{Z}$.

- If $k > 0$, then $\overline{a}^k := \overline{a} \cdot \overline{a} \cdot \ldots \cdot \overline{a}$ with exactly k factors.
- If $k = 0$, then $\overline{a}^k := \overline{1}$.
- If $k < 0$ and $\overline{a}$ is invertible, then $\overline{a}^k := \left(\overline{a}^{-1}\right)^{-k}$ (note: $-k$ is positive).

For example, in $\mathbb{Z}_7$, we know that the inverse of $\overline{5}$ is $\overline{3}$, so $\overline{5}^{-3} = \left(\overline{5}^{-1}\right)^3 = \overline{3}^3 = \overline{3} \cdot \overline{3} \cdot \overline{3} = \overline{2} \cdot \overline{3} = \overline{6}$.

Note: the notation for inverse, $\overline{a}^{-1}$ coincides with negative exponentiation for $k = -1$.

The following theorem ensures that exponentiation is valid for residue classes, i.e. the result doesn't depend on which element of the residue class was exponentiated:

Theorem 2.2. *If $a \equiv b \pmod{m}$, then $a^k \equiv b^k \pmod{m}$.*

For example, $5^3 \equiv 12^3 \pmod 7$, i.e. it doesn't matter whether we use the $\overline{5}^3$ or the $\overline{12}^3$ form, the result will be the same. (Warning: this is only true for the base, not for the exponent! For example, $5^{10} \not\equiv 5^3 \pmod 7$. We'll discuss this later.)

In Sage, there are multiple ways to exponentiate modularly:

```
sage: 3^100 % 7              # SLOW! (it first calculates 3^100 as an integer)
4
sage: Z7 = Zmod(7)
sage: Z7(3)^100              # returns a residue class
4
sage: power_mod(3, 100, 7)  # returns an integer
4
```

Now calculate the first few powers of all elements in $\mathbb{Z}_7^*$:

```
sage: matrix([[a^k for k in range(20)] for a in Zmod(7) if a != 0])
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 2 4 1 2 4 1 2 4 1 2 4 1 2 4 1 2 4 1 2]
[1 3 2 6 4 5 1 3 2 6 4 5 1 3 2 6 4 5 1 3]
[1 4 2 1 4 2 1 4 2 1 4 2 1 4 2 1 4 2 1 4]
[1 5 4 6 2 3 1 5 4 6 2 3 1 5 4 6 2 3 1 5]
[1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6]
```

First notice that the powers repeat periodically. Of course they repeat, because $\mathbb{Z}_7^*$ has only finitely many elements, so the powers can't be all different. We can also notice that the first element to repeat is always $\overline{1} = \overline{a}^0$. Why is that so? Because if we have any repetition, e.g. $\overline{a}^8 = \overline{a}^2$, then we can divide it by $\overline{a}^2$, and we get an earlier repetition: $\overline{a}^6 = \overline{1}$. Now let's find out *when* this repeation happens first.

Definition 2.3. The *order* of an element $\overline{a} \in \mathbb{Z}_m$ is the smallest $k \in \mathbb{N}^+$ for which $\overline{a}^k = \overline{1}$.

Notation: $o(\overline{a})$ or $\mathrm{ord}(\overline{a})$.

So the order is the smallest exponent for which the power becomes $\overline{1}$. Therefore, it is also the length of the period of the repetition (since $\overline{a}^{j+o(\overline{a})} = \overline{a}^j \overline{a}^{o(\overline{a})} = \overline{a}^j \overline{1} = \overline{a}^j$).

For example, in $\mathbb{Z}_7$, the order of $\overline{4}$ is 3 ($o(\overline{4}) = 3$), because $\overline{4}^3 = \overline{1}$, but $\overline{4}^1 = \overline{4} \neq \overline{1}$ and $\overline{4}^2 = \overline{2} \neq \overline{1}$. We can also see from the above table that $o(\overline{5}) = 6$.

From the table, we can also see that for each element in $\mathbb{Z}_7^*$, their sixth power is $\overline{1}$ (but this is not always the first $\overline{1}$). This is true in general as well:

Theorem 2.4 (Fermat's little theorem)**.** *If p is prime, $a \in \mathbb{Z}$ and $p \nmid a$, then $a^{p-1} \equiv 1 \pmod p$.*

The same with residue classes: $\forall \overline{a} \in \mathbb{Z}_p^* : \overline{a}^{p-1} = \overline{1}$.

So the order of any element can be at most $p-1$ (i.e. $o(\overline{a}) \leq p-1$). But we can say something more specific about the order:

Theorem 2.5 (Lagrange's theorem)**.** *If p is a prime and $\overline{a} \in \mathbb{Z}_p^*$, then $o(\overline{a}) \mid p-1$.*

Indeed, we can see from the above table that the orders of the elements of $\mathbb{Z}_7^*$ are 1, 2, 3 and 6, which are divisors of $7-1 = 6$.

Let's emphasize two special cases. First, of course $o(\overline{1}) = 1$ for any modulus, since any power of $\overline{1}$ is $\overline{1}$. Then, we can see that $o(\overline{6}) = 2$ in $\mathbb{Z}_7$, and this is also true in general: in $\mathbb{Z}_p$, $o(\overline{p-1}) = 2$, because $\overline{p-1}^2 = \overline{-1}^2 = \overline{1}$ (except in $\mathbb{Z}_2$, where $\overline{-1} = \overline{1}$).

Also notice that for each element in $\mathbb{Z}_p^*$, we can find its inverse in the list of its powers. Why? Because we know that $\overline{a}^{p-1} = \overline{1}$, and if we divide it by $\overline{a}$, we get $\overline{a}^{p-2} = \overline{a}^{-1}$. The first occurance of $\overline{a}^{-1}$ is $\overline{a}^{-1} = \overline{a}^{o(\overline{a})-1}$. (For example, in $\mathbb{Z}_7$, the inverse of $\overline{4}$ is $\overline{2}$, its order is 3, and indeed: $\overline{4}^{3-1} = \overline{2}$.)

3 Fast exponentiation

Now let's see how we can calculate $\overline{a}^n$ efficiently in a given $\mathbb{Z}_p$ structure.
First, Fermat's little theorem says that $\overline{a}^{p-1} = \overline{1}$, so instead of the exponent n, we can use its remainder by $p-1$: $\overline{a}^n = \overline{a}^{n \bmod p-1}$. For example, in $\mathbb{Z}_7$, $\overline{5}^6 = \overline{1}$, so any $\overline{5}^{6k}$ is also $\overline{1}$, therefore e.g. $\overline{5}^{63} = \overline{5}^{63 \bmod 6} = \overline{5}^3 = \overline{6}$. (Remember the warning that we can't calculate modularly with the exponent in the same way as with the base. Well, not *in the same way*, but we can do it with another modulus $(p-1)$, e.g. in $\mathbb{Z}_7$, with 6. We'll discuss this later more.)
But this is not enough, because the reduced exponent can still be very large if the modulus is big. Therefore, we introduce an algorithm called *fast exponentiation* (also known as *exponentiation by squaring*). This works not only in $\mathbb{Z}_p$, but in any structure that has exponentiation (e.g. integers, real numbers, matrices etc.), and we need to calculate it with as few multiplications as possible.
For example, how can we calculate a^8? By definition, it is $a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a$. But this involves seven multiplications. Can we do this with fewer multiplications? Yes, we can: $a^8 = \left(a^4\right)^2 = \left(\left(a^2\right)^2\right)^2$, which requires only three multiplications (three squarings). But what if the number is not a power of two? If it's even, then the first step is the same: $a^{2n} = (a^n)^2$, and we can continue it as long as the inner n remains even; and if it's odd, we can seperate an a: $a^{2n+1} = (a^n)^2 \cdot a$, and continue in the same way. For example:

$$a^{22} = \left(a^{11}\right)^2 = \left(\left(a^5\right)^2 \cdot a\right)^2 = \left(\left(\left(a^2\right)^2 \cdot a\right)^2 \cdot a\right)^2.$$

In this way, calculating a^{22} requires not 21, but only 6 multiplications.
Notice that the above representation of a^{22} has a strong connection with the exponent's binary representation. The sequence of operations consists of a squaring in each step, optionally followed by a multiplication by a. If we write a 1 each time a multiplication by a happens, a 0 each time when it didn't happen after a squaring, and prefix the whole string by a 1 (for the innermost a), then we get the binary representation of the original exponent, e.g. $22 = 10110_2$.
The complete algorithm of fast exponentiation is as follows:

1. Input: a (any exponentiable value) and the exponent $n \in \mathbb{N}^+$.
2. Write n in binary (base-2) form: $n = (b_l \dots b_1 b_0)_2$, i.e. $n = b_0 + b_1 2 + \dots b_l 2^l$ (where $b_l = 1$).
3. $r := a$.
4. For each i from $l-1$ to 0:
   (a) $r := r \cdot r$.
   (b) If $b_i = 1$, then $r := r \cdot a$.
5. Output: r.

This algorithm is very efficient, because it makes at most $2l$ multiplications, where l is approximately the length of the exponent in binary, i.e. $l \approx \log_2 n$. For example, if $n = 1000000$, then it performs at most 40 multiplications.

4 Discrete logarithm

We have seen that the order of any element in $\mathbb{Z}_p^*$ is at most $p-1$. Now consider those elements whose order is the largest possible value:

Definition 4.1. For a prime p, an element $\overline{g} \in \mathbb{Z}_p^*$ is a *generator* in $\mathbb{Z}_p^*$ if $o(\overline{g}) = p-1$.

For example, $\overline{3}$ and $\overline{5}$ are generators in $\mathbb{Z}_7$, because their order is 6 (see the table of powers above).

But why are they called "generators"? Let's list the powers of $\overline{5}$ again:

```
sage: [mod(5, 7)^k for k in range(0, 20)]
[1, 5, 4, 6, 2, 3, 1, 5, 4, 6, 2, 3, 1, 5, 4, 6, 2, 3, 1, 5]
```

Notice that this list contains all elements of $\mathbb{Z}_7^*$. This is true in general:

Theorem 4.2. *If $\overline{g} \in \mathbb{Z}_p^*$ is a generator, then $\forall \overline{a} \in \mathbb{Z}_p^* : \exists k \in \mathbb{N} : \overline{g}^k = \overline{a}$.*

In other words: a generator generates all elements of $\mathbb{Z}_p^*$ by its powers.

(The proof is simple: the order of the generator is $p-1$ by definition, so in the list of its powers, the first repetition happens after $p-1$ elements, which means that the first $p-1$ elements are all different; but $\mathbb{Z}_p^*$ has exactly $p-1$ elements, so all of them must be present in the list.)

Theorem 4.3. *If p is prime, then there exists a generator in $\mathbb{Z}_p^*$.*

(The proof is difficult, so it is omitted.)

So generators always exist (for a prime modulus), and they generate all nonzero elements. Also, each element is generated only once within the first $p-1$ exponents. So the following question naturally arises: where can we find a given element in the list? This question is the inverse of exponentiation, which asks for the value of the power from the base and the exponent; now we ask for the exponent if we know the base and the power. This is very similar to what logarithm does for real numbers: the real equation $b^x = a$ has the solution $x = \log_b a$, e.g. the solution of $2^x = 4\sqrt{2}$ is $x = \log_2 4\sqrt{2} = 2.5$.

Now let's see the modular analogue of logarithm:

Definition 4.4. If $\overline{a}, \overline{b} \in \mathbb{Z}_p^*$, then the smallest $x \in \mathbb{N}$ for which $\overline{b}^x = \overline{a}$ is called the *discrete logarithm* of $\overline{a}$ to the base $\overline{b}$ (or the *index* of $\overline{a}$). Notation: $x = \log_{\overline{b}} \overline{a}$ (or $x = \mathrm{ind}_{\overline{b}}\, \overline{a}$).

For example, in $\mathbb{Z}_7$, the discrete logarithm of $\overline{6}$ to the base $\overline{5}$ is 3, because $\overline{5}^3 = \overline{6}$ (see the powers of $\overline{5}$ above).

If $\overline{g} \in \mathbb{Z}_p^*$ is a generator, then it guarantees that $\log_{\overline{g}} \overline{a}$ exists for all $\overline{a} \in \mathbb{Z}_p^*$.

But how can we calculate the discrete logarithm? The simplest way is to calculate all powers of $\overline{g}$ from 0 to $p-2$ until one of them matches $\overline{a}$, and then its exponent is the discrete logarithm. But for large p, this is very slow, because we need at most $p-1$ multiplications, e.g. if $p \approx 1000000$, this means one million multiplications in the worst case.

So let's see a slightly more efficient method, the *baby-step giant-step* algorithm. The idea is to pick a number N in the middle, i.e. for which $N^2 \approx p-1$, and write the unknown exponent of $\overline{a} = \overline{g}^x$ in the form $x = iN + j$, where $0 \le i, j < N$ (e.g. if $N = 10$ and $x = 76$, then $x = 7N + 6$, i.e. $i = 7$ and $j = 6$). In the first step, calculate the first few $\overline{g}^j$ powers: $\overline{1}$, $\overline{g}$, $\overline{g}^2$, $\ldots$, $\overline{g}^{N-1}$ ("baby-step"), and store them. Then start from the given $\overline{a} = \overline{g}^x$ element, and multiply it repeatedly by $\overline{g}^{-N}$: $\overline{a}$, $\overline{g}^{-N}\overline{a}$, $\overline{g}^{-2N}\overline{a}$, $\ldots$ ("giant-step"). The point of this is that the two sequences (the "baby-step" and the "giant-step") will meet: when $\overline{a}$ is multiplied by exactly $\overline{g}^{-iN}$ (where i is from the form $x = iN + j$, i.e. it is yet unknown), then $\overline{g}^{-iN}\overline{a} = \overline{g}^{-iN+x} = \overline{g}^{-iN+(iN+j)} = \overline{g}^j$, i.e. we can find it among the stored $\overline{g}^j$ values. When it happens, we have found i and j, and so x.

The complete baby-step giant-step algorithm is as follows:

1. Input: p prime, $\overline{a}, \overline{g} \in \mathbb{Z}_p^*$, where $\overline{g}$ is a generator.
2. $N := \left\lceil \sqrt{p-1} \right\rceil$.
3. $\overline{b} := \overline{1}$.
4. For each j from 0 to $N-1$:
   (a) Store $(\overline{b}, j)$ in a table. (Note: $\overline{b} = \overline{g}^j$.)
   (b) $\overline{b} := \overline{b} \cdot \overline{g}$.
5. $\overline{c} := \overline{b}^{-1}$ $(= \overline{g}^{-N})$.
6. $\overline{b} := \overline{a}$.
7. For each i from 0 to $N-1$:
   (a) If $\overline{b}$ is stored in the table for some j, then stop. The result is: $x := iN + j$.
   (b) $\overline{b} := \overline{b} \cdot \overline{c}$.

In order for this algorithm to be efficient, our "table" must store and find elements very quickly. For example, we can use a hash table, which stores and finds elements in constant time (i.e. the time doesn't depend on the size of the table). The key of the table is $\overline{b}$, and the value is the index j. In Sage, we can use Python's dictionary (`dict`) type for this (empty dict: `T = {}`, insert an element: `T[b] = j`, find an element: `b in T`, access that element: `T[b]`).

The above algorithm performs ca. $2\sqrt{p}$ multiplications, which is much better than the first naive approach (e.g. for $p \approx 1000000$, it does ~ 2000 multiplications). But if we compare this to the running time of fast exponentiation, then we can see that this is still slow for large p. Also, it requires much more memory too (it needs to store ca. $2\sqrt{p}$ numbers).

So how can we calculate the discrete logarithm really efficiently, just like we did for exponentiation? The answer is: we don't know. So much so, that the discrete logarithm problem is still an open question of mathematics today. But this is not a problem. In fact, as we'll see, this is good news!

In Sage, the same `log(a, b)` function that calculates the real logarithm works for discrete logarithm too, if it gets residue classes (but of course it won't work for very large moduli):

```
sage: Z7 = Zmod(7)
sage: log(Z7(6), Z7(5))
3
sage: Z7(5)^3
6
```

5 Diffie–Hellman key exchange

Consider two people who would like to agree on a secret information. But the problem is that they can't talk privately, only in a public place where anything they say could be overheard by someone else. How can they still establish a shared secret between each other?

The question is not theoretic. Think about the Internet, where non-encrypted messages can be easily intercepted. But how can two parties start an encrypted communication if they haven't met before, i.e. how can they establish a common secret key, necessary for encryption, if they can only communicate on the public Internet?

This seemingly impossible problem is solved by the *Diffie–Hellman key exchange*. This method is based on the fact that the discrete logarithm problem is hard, i.e. if we have a generator $\overline{g} \in \mathbb{Z}_p^*$ and a secret exponent $n \in \mathbb{N}$, then we can share $\overline{g}^n$ publicly, because nobody will be able to calculate n. (At least if p is large enough, and it satisfies certain other mathematical requirements, which are not detailed here.)

The Diffie–Hellman key exchange works as follows:

1. The two participants, Alice and Bob agree on a prime number p and a generator $\overline{g} \in \mathbb{Z}_p^*$ (these values can be shared publicly, or they can even be fixed constants).
2. They both generate their own random secret exponent: $a, b \in \mathbb{N}$, where a is only known by Alice, and b is only known by Bob.
3. Alice calculates $\overline{g}^a$, and sends it to Bob.
4. Bob calcultes $\overline{g}^b$, and sends it to Alice.
5. Then they both calculate $\overline{g}^{ab}$ from the information they know:
   (a) Alice knows a and $\overline{g}^b$, so she calculates $\overline{g}^{ab} = \left(\overline{g}^b\right)^a$.
   (b) Bob knows b and $\overline{g}^a$, so he calculates $\overline{g}^{ab} = \left(\overline{g}^a\right)^b$.

The common secret information is $\overline{g}^{ab}$, which both Alice and Bob can calculate with the above steps. But anyone else who listens on the channel can only see $\overline{g}$, $\overline{g}^a$ and $\overline{g}^b$, and they can't calculate $\overline{g}^{ab}$ from these values, because they would have to solve the discrete logarithm problem for that.

Note: in order for this method to be really secure, certain conditions must be met. First, as mentioned above, p must be large enough and appropriately chosen. But also, the random secret exponents, a and b, must be *random enough* so that an outside attacker can't predict them (with certain probability), otherwise the whole algorithm is useless.

Cryptographically secure pseudorandom number generators (CSPRNG) have certain mathematical guarantees to ensure that an outside listener can't predict the subsequent numbers. Non-secure random number generators may be simpler and more efficient, but they lack these guarantees, and therefore they shouldn't be used if security is important (e.g. in the Diffie–Hellman key exchange). Like most programming langauges, Python (and therefore Sage) has both secure and non-secure random number generators:

```
sage: randrange(100)           # NOT secure (between 0 and 99)
94
sage: randrange(100, 200, 10)  # expects the same parameters as range()
180
sage: import secrets
sage: secrets.randbelow(100)   # secure (between 0 and 99)
67
```

6 Author

These lecture notes were written by Uray M. János, partially using Nagy Ádám's notes in Hungarian:
https://compalg.elte.gitlab-pages.hu/dimoa-web/tematika/dimoa
Please report errors and suggestions here: uray.janos@inf.elte.hu.