Discrete logarithm

Application of discrete models

1 Modular exponentiation

In the previous document (Congruences), we stated the following property about exponentiation:

Theorem 1.1. *If* $a \equiv b \pmod{m}$*, then* $\forall n \in \mathbb{N} : a^n \equiv b^n \pmod{m}$.

Example. $5 \equiv 12 \pmod{7} \implies 5^3 \equiv 12^3 \pmod{7}$, i.e. it doesn't matter which one of the congruent bases (5 and 12) are used for exponentiating. So it is worth taking the remainder of the base first.

(Warning: this is only true for the base, not for the exponent! E.g. $5^{10} \not\equiv 5^3 \ (7)$. This will be discussed later.)

For small exponents, we can perform modular exponentiation by first calculating the power as an integer, and then taking its remainder. For example, for $5^5 \bmod 7$:

$$5^5 = 5 \cdot 5 \cdot 5 \cdot 5 \cdot 5 = 3125 \implies 5^5 \equiv 3125 \equiv 3 \pmod{7}.$$

For bigger exponents however, this can result in prohibitively large numbers. Therefore, it is better to take remainder after *each* multiplication, because then we can keep the intermediate results small:

$$\begin{aligned} 5^2 &\equiv 5 \cdot 5 \equiv 25 \equiv 4 \pmod{7}, \\ 5^3 &\equiv 5^2 \cdot 5 \equiv 4 \cdot 5 \equiv 20 \equiv 6 \pmod{7}, \\ 5^4 &\equiv 5^3 \cdot 5 \equiv 6 \cdot 5 \equiv 30 \equiv 2 \pmod{7}, \\ 5^5 &\equiv 5^4 \cdot 5 \equiv 2 \cdot 5 \equiv 10 \equiv 3 \pmod{7}. \end{aligned}$$

Similarly in Sage, the naive calculation, `a^n % m`, should be avoided, because it first calculates `a^n`, which can be very big. Instead, there is a built-in function that does it much more efficiently:

```
sage: 5^999999999 % 7              # SLOW!
6
sage: power_mod(5, 999999999, 7)   # fast
6
```

In this document, we focus on the special case of modular exponentiation when the modulus is a prime number. (The general case will be discussed in the next document.)

This assumption simplifies certain operations. For example, we have seen in the previous document that in this case, every nonzero remainder is invertible. The multiplication also works better if the modulus is prime:

Theorem 1.2. *If* p *is prime,* $a \not\equiv 0 \ (p)$ *and* $b \not\equiv 0 \ (p)$*, then* $a \cdot b \not\equiv 0 \ (p)$.

That is, the product of nonzero remainders never gives a zero remainder. This corresponds to the fact that the product of two nonzero real numbers is never zero. It seems very natural, and it will be very useful, but this is only true if the modulus is prime! For example, if the modulus is $m = 10$, then $4 \cdot 5 \equiv 20 \equiv 0 \pmod{10}$. (Later such numbers will be called *zero divisors*.)

And since the exponentiation is just repeated multiplication, powers of nonzero remainders are also nonzero:

Theorem 1.3. *If* p *is prime and* $a \not\equiv 0 \ (p)$*, then* $\forall n \in \mathbb{N} : a^n \not\equiv 0 \ (p)$.

(But again, this is only true for a prime modulus, otherwise e.g. $6^2 \equiv 0 \pmod{12}$.)

2 Fast exponentiation

In this section, we describe a very efficient method for calculating the modular power $a^n \bmod m$: the **fast exponentiation** (or *exponentiation by squaring*). (It is also used by Sage's `power_mod()`.)

First, let's ignore the modulus. For example, how can we calculate a^8? The naive method is $a^8 = a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a \cdot a$, which is 7 multiplications. Can we do better? Yes: $a^8 = \left(a^4\right)^2 = \left(\left(a^2\right)^2\right)^2$, which is only 3 multiplications (3 squarings). This can be generalized as follows: if the exponent is 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 is odd, then we can first seperate an a: $a^{2n+1} = (a^n)^2 \cdot a$, and then 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 to the binary form of 22. In each step, we perform a squaring, and optionally multiply the result by a. If we write '1' each time this multiplication happens, and a '0' otherwise, and prefix the whole string by a '1' (for the innermost a), then we get the binary representation of the exponent, e.g. $22 = 10110_2$.

Since this is modular exponentiation, we need to take the remainder – and after each step, as we have seen above, otherwise the number could grow very big, making "fast exponentiation" slow.

Fast exponentiation(a, n, m)**:**

Input: $a \in \mathbb{Z}$ (the base), $n \in \mathbb{N}^+$ (the exponent), $m \in \mathbb{N}^+$ (the modulus)
$B :=$ Binary representation(n) *(see below)*
$a := a \bmod m$
$r := a$
for $b :=$ *elements of* B *except the first (which is* 1*)* **do**
  $r := r \cdot r \bmod m$
  if $b = 1$ **then**
    $r := r \cdot a \bmod m$
Output: r

Binary representation(n)**:**

Input: $n \in \mathbb{N}^+$
$B :=$ *(empty list)*
while $n \neq 0$ **do**
  Add $(n \bmod 2)$ to the end of B
  $n := n \operatorname{div} 2$
Output: B in reverse order

Example. Binary representation$(8) = [1, 0, 0, 0]$ and Binary representation$(22) = [1, 0, 1, 1, 0]$.

This algorithm is very efficient, because it does at most $2L$ multiplications, where L is the binary length of the exponent, i.e. $L \approx \log_2 n$. For $n \approx 1000000$, this means at most $\sim$40 multiplications.

Note: fast exponentation works not only for modular exponentation, but also for exponentiating any kind of mathematical object that supports multiplication (e.g. integer, real number, matrix etc.). Then the remainder calculations (mod m) are obviously omitted.

3 Structure of the powers

Now let's examine the modular powers of various numbers modulo 7:

```
sage: matrix([[power_mod(a, n, 7) for n in range(30)] for a in range(1, 7)])
[1 1 1 1 1 1 1 1 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 4 1 2 4 1 2 4 1]
[1 3 2 6 4 5 1 3 2 6 4 5 1 3 2 6 4 5 1 3 2 6 4 5 1 3 2 6]
[1 4 2 1 4 2 1 4 2 1 4 2 1 4 2 1 4 2 1 4 2 1 4 2 1 4 2 1]
[1 5 4 6 2 3 1 5 4 6 2 3 1 5 4 6 2 3 1 5 4 6 2 3 1 5 4 6]
[1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6 1 6]
```

Notice that the powers repeat periodically. (Indeed, there are only finitely many remainders, so the powers cannot be all different.) Also, the first repeating power is always 1.

But why exactly 1? Because if we have any other repetition, e.g. $a^8 \equiv a^2 \pmod 7$, then we can divide it by a^2 – more precisely multiply it by the modular inverse of a^2 –, and get an earlier repetition: $a^6 \equiv 1 \pmod 7$.

But *when* does this repeation happen first?

Definition 3.1. The **order** of $a \in \mathbb{Z}$ modulo m is the smallest exponent $n \in \mathbb{N}^+$ for which $a^n \equiv 1 \pmod m$. Notation: $\mathrm{ord}_m(a)$.

Example. $\mathrm{ord}_7(4) = 3$ (the order of 4 mod 7 is 3), because $4^3 \equiv 1\ (7)$, but $4^2 \not\equiv 1\ (7)$ and $4^1 \not\equiv 1\ (7)$.

Example. $\mathrm{ord}_7(5) = 6$, as we can see from the above table.

So the order is the smallest positive exponent for which the modular power is 1. But since 1 is the first repeated power, it means that the order is also the length of the period of the repetition:

$$a^{n+\mathrm{ord}_m(a)} \equiv a^n a^{\mathrm{ord}_m(a)} \equiv a^n \cdot 1 \equiv a^n \pmod m.$$

From the above table, we can also see that the sixth power of each nonzero remainder is 1 (but this is not always the first 1). This can be generalized as follows:

Theorem 3.2 (Fermat's little theorem)**.** *If p is prime and $a \not\equiv 0 \pmod p$, then*

$$a^{p-1} \equiv 1 \pmod p.$$

So the order of any number is at most $p-1$, i.e. $\mathrm{ord}_p(a) \le p-1$. More specifically:

Theorem 3.3 (Lagrange's theorem)**.** *If p is prime and $a \not\equiv 0 \pmod p$, then* $\mathrm{ord}_p(a) \mid p-1$.

Indeed, we can see that the order of the numbers are 1, 2, 3 or 6, which are divisors of $7-1 = 6$.

There are two special cases of note. First, of course $\mathrm{ord}_m(1) = 1$ for any modulus, because any power of 1 is 1. Second, we can see that $\mathrm{ord}_7(6) = 2$, which can be generalized as follows: $\mathrm{ord}_m(m-1) = 2$, because $m-1$ is congruent to -1, and $(-1)^2 = 1$, i.e. $(m-1)^2 \equiv (-1)^2 \equiv 1 \pmod m$ (an exception is $m = 2$, when this is not the first 1, because $-1 \equiv 1 \pmod 2$).

It is also worth noting that the the modular inverse of any number can be found in the list of its powers, right before the 1. Why? Because Fermat's little theorem says that $a^{p-1} \equiv 1\ (p)$, and multiplying this by a^{-1} gives $a^{p-2} \equiv a^{-1}\ (p)$. The first occurance of a^{-1} is $a^{-1} \equiv a^{\mathrm{ord}_p(a)-1}\ (p)$. (E.g. for $p = 7$, the inverse of 4 is 2, its order is 3, and indeed: $4^{3-1} \equiv 2\ (7)$.) This gives a new method for finding the modular inverse (in addition to the ones described in the previous document).

Now consider the numbers whose order is the largest possible value:

Definition 3.4. For a prime p, the integer $g \in \mathbb{Z}$ is a **primitive root** modulo p if $\mathrm{ord}_p(g) = p-1$. (It is also called a **generator**.)

Example. 3 and 5 are primitive roots modulo 7, because their order is 6 (see the above table).

Why are primitive roots interesting? For example, the powers of 5 modulo 7 are the following: $1, 5, 4, 6, 2, 3, 1, 5, \ldots$. Notice that this list contains every nonzero remainder. This is true in general:

Theorem 3.5. *If p is prime and g is a primitive root modulo p, then $\forall b \not\equiv 0\ (p) : \exists n \in \mathbb{N} : g^n \equiv b\ (p)$.*

In other words: a primitive root generates every possible nonzero remainder by its powers.

(The proof is simple: the order of the primitive root is by definition $p-1$, so the first repetition in the list happens after $p-1$ powers, which means that the first $p-1$ powers are all different; but there are exactly $p-1$ remainders, so all of them must be present in the list.)

Do primitive roots exist for all prime numbers? The answer is yes (but it is difficult to prove):

Theorem 3.6. *If p is prime, then there exists a primitive root modulo p.*

4 Discrete logarithm

Now examine the inverse operation of modular exponentiation: given a power and a base, which exponent gives that power? This is very similar to what logarithm does for real numbers: the solution of the real equation $a^x = b$ is given by $x = \log_a b$, e.g. the solution of $2^x = 256$ is $x = \log_2 256 = 8$.

Definition 4.1. The **discrete logarithm** (or **index**) of $b \in \mathbb{Z}$ to the base $a \in \mathbb{Z}$ modulo m is the smallest exponent $n \in \mathbb{N}$ for which $a^n \equiv b \pmod{m}$. Notation: $x = \operatorname{ind}_a b \pmod{m}$.

Example. $\operatorname{ind}_5 6 = 3 \pmod 7$, i.e. the discrete logarithm of 6 to the base 5 modulo 7 is 3, because $5^3 \equiv 6\ (7)$, and no earlier power of 5 is 6 (the first few powers are: $1, 5, 4, 6, \ldots$).

Example. What is $\operatorname{ind}_4 3 \pmod 7$, i.e. the base-4 discrete logarithm of 3 modulo 7? The powers of 4 are $1, 4, 2, 1, 4, 2, 1, \ldots$, and none of them is 3, so $\operatorname{ind}_4 3 \pmod 7$ does not exist.

If g is a primitive root modulo p, then by Theorem 3.5. above, $\operatorname{ind}_g b$ exists for all $b \not\equiv 0\ (p)$.

The simplest way to calculate the discrete logarithm is to generate the powers $1, a, a^2, a^3, \ldots \pmod m$, and if one of them matches b, then its exponent will be the result. If none of them do, then we can stop when the next power is 1, because then the numbers will only repeat. It is not necessary to calculate the powers independently, because each one can be calculated from the previous one by a single multiplication and a remainder calculation. Fermat's little theorem ensures that this algorithm takes at most $p-1$ steps, i.e. it is *linear*. This means that it is very slow for large p, for example, for $p \approx 1000000$, it takes around one million steps in the worst case.

There are *somewhat* more efficient algorithms than the above linear search (such as the one described in the next section), but none of them gets anywhere near the fast exponentiation in speed. That is, we can do modular exponentiation efficiently, but not its inverse, the discrete logarithm – which is still an open question of mathematics today.

But this is not a problem. In fact, as we will see, this is really useful!

Sage can calculate the discrete logarithm using the `discrete_log()` function (which is much more general than what we need, so it's a bit cumbersome to use):

```
sage: discrete_log(mod(6, 7), mod(5, 7))   # base-5 log 6 (mod 7)
3
sage: power_mod(5, 3, 7)    # check
6
```

But of course, if p is very big, `discrete_log()` can't solve it either:

```
sage: p = next_prime(2^1000)
sage: b = power_mod(7, randrange(p), p)
sage: discrete_log(mod(b, p), mod(7, p))    # won't work
...
```

Warning: don't confuse `mod(b, p)` with `(b % p)`, because they return different types (more on this in a later document). But `discrete_log()` requires exactly what `mod(b, p)` returns.

4.1 Baby-step giant-step

Here is a discrete logarithm algorithm that is faster than the above linear search. The idea is to take a number M in the middle, i.e. for which $M^2 \approx p-1$, and write the unknown exponent of $a^n \equiv b\ (p)$ in the form $n = x+My$, where $0 \leq x, y < M$ (e.g. if $M = 10$ and $n = 76$, then $n = 7M+6$, i.e. $x = 6$ and $y = 7$). First, calculate the first few a^x powers (modulo p): $1,\ a,\ a^2,\ \ldots,\ a^{M-1}$ ("baby-step"), and store them. Then, start from the known power b, and multiply it repeatedly by a^{-M}, i.e.: $b,\ a^{-M}b,\ a^{-2M}b,\ \ldots$ ("giant-step"). These two sequences (the "baby-step" and the "giant-step") will eventually meet: when b is multiplied by exactly a^{-yM} (where y is an unknown from the exponent $n = x + My$), then $a^{-yM}b = a^{-yM+n} = a^{-yM+(x+My)} = a^x$, which is among the stored a^x values. When it happens, we have x and y, and so we can construct n from them.

Baby-step giant-step algorithm:

Input: p prime, $a \in \{1, \ldots, p-1\}$ (the base), $b \in \{1, \ldots, p-1\}$ (the power)
$M := \left\lceil \sqrt{p-1} \right\rceil$
$c := 1$
for $x = 0, 1, \ldots, M-1$ **do**
  Store (c, x) in a table. (Note: $c \equiv a^x\ (p)$.)
  $c := c \cdot a \bmod p$
$d := c^{-1} \bmod p\ \ (= a^{-M} \bmod p)$
$c := b$
for $y = 0, 1, \ldots, M-1$ **do**
  if *c is in the table for some x* **then**
    the result is: $\rightarrow n := x + My$
  $c := c \cdot d \bmod p$

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. The key of the table is c, and the value is the exponent x. In Sage, we can use Python's dictionary (`dict`) type for this (empty dict: `T = {}`, insert an element: `T[c] = x`, find an element: `c in T`, access that element: `T[c]`).

The algorithm performs $\sim 2\sqrt{p}$ multiplications, which is much better than the linear search (e.g. for $p \approx 1000000$, it does ~ 2000 multiplications). But if we compare it to fast exponentiation, then it is still slow for large p. Also, it requires much more memory too (it stores $\sim 2\sqrt{p}$ numbers).

5 Diffie–Hellman key exchange

Assume that two people would like to communicate privately through a public channel (e.g. the internet). So they use encryption, but for that, they need a *secret key* that allows them to read each other's encrypted messages. The problem is that they have never met, so they can only communicate the secret key through the same public channel, where anything they say can be overheard by someone else. How can they still establish the shared secret key between each other?

This seemingly impossible problem can be 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 secret exponent $n \in \mathbb{N}$, then we can share the modular power $g^n \bmod p$ publicly (along with g and p), because nobody will be able to calculate the secret exponent n back from them.

The Diffie–Hellman key exchange works as follows (every exponentiation is meant modulo p):

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

So the common secret is g^{ab}, which both Alice and Bob can calculate with the above steps. But anyone else who listens on the channel can only see g, g^a and g^b, from which they cannot calculate g^{ab}, because they also need one of the secret exponents (a or b), and for that, they would need to solve the discrete logarithm problem.

Example.

1. Let $p = 23$ and the primitive root $g = 5$.
2. Alice generates the secret exponent $a = \mathbf{7}$, and Bob generates $b = \mathbf{21}$.
3. Alice calculates $g^a \equiv 5^7 \equiv \mathbf{17} \pmod{23}$, and sends it to Bob.
4. Bob calculates $g^b \equiv 5^{21} \equiv \mathbf{14} \pmod{23}$, and sends it to Alice.
5. They both can calculate g^{ab}:
   - Alice: $g^{ab} \equiv \left(g^b\right)^a \equiv 14^7 \equiv \mathbf{19} \pmod{23}$.
   - Bob: $g^{ab} \equiv \left(g^a\right)^b \equiv 17^{21} \equiv \mathbf{19} \pmod{23}$.

In order for the Diffie–Hellman key exchange to be secure, the prime p must be large enough. In practice, p is usually several thousand bits long (e.g. 2048 bits long, which means that its binary representation has 2048 digits, which is 617 digits in decimal), see e.g.: [5].

Warning: the above description merely illustrates the main points of the algorithm, but it contains simplifications, and it is not sufficient for a secure implementation of the key exchange. In general, implementing cryptographic algorithms – other than toy examples – should be left to experts in the field, because even the smallest mistakes can cause serious security flaws.

The Diffie–Hellman key exchange is used in various places, including internet systems such as HTTPS, SSH, VPNs etc. – usually in combination with other cryptographic algorithms like RSA and AES (which will be discussed in later documents).

6 Random number generators

Most programming languages have a *random number generator (RNG)*. But most of them don't generate "true" random numbers (which can be created e.g. by throwing dice or by other physical means), instead they calculate the numbers using a simple algorithm, which produces numbers that look "random enough". Such numbers are called *pseudorandom numbers*.

A very simple pseudorandom number generator (PRNG) can be created using modular exponentiation. Although the powers repeat periodically, they look quite random within one period (e.g. the powers of 5 modulo 7 are: $1, 5, 4, 6, 2, 3$). For example, C++'s `std::minstd_rand` [6] generator uses the following parameters:

$$x_n := 48271^n x_0 \bmod 2147483647,$$

where x_0 is an arbitrary initial value called the *seed* (which is often calculated from the current time, to make it always different); the modulus $p = 2147483647 = 2^{31} - 1$ is a prime number that was chosen so that the remainders just fit into a signed 32-bit integer (and that the remainder calculation can be done efficiently); and the base, 48271 is a primitive root modulo p.

This random number generator is very simple and very efficient, so it is well-suited e.g. for simulations and games. However, it is not suitable for generating the exponents of the Diffie–Hellman key exchange, a and b (nor in any other algorithm where security is important), because its simplicity means that it is very easy to predict the next numbers, which makes the whole key exchange useless.

Instead, *cryptographically secure pseudorandom number generators* (CSPRNG) should be used, which are more complex, but they have sufficient mathematical guarantees that an outside listener can't predict the subsequent numbers (not even some of its digits, not even with some probability).

Many programming langauges, including Sage (and Python) have both secure and non-secure random number generators:

```
sage: randrange(100)                # NOT secure
94
sage: import secrets
sage: secrets.randbelow(100)        # secure
67
sage: secrets.randbelow(int(100))  # same, but for older Sage versions
56
```

7 Author

This document was written by Uray M. János, partly using the following sources:

[1] https://compalg.elte.gitlab-pages.hu/dimoa-web/tematika/dimoa (Hungarian)

[2] https://en.wikipedia.org/wiki/Modular_exponentiation

[3] https://en.wikipedia.org/wiki/Discrete_logarithm

[4] https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange

[5] Primes for the Diffie–Hellman key exchange: https://www.rfc-editor.org/rfc/rfc3526

[6] https://en.cppreference.com/w/cpp/numeric/random/linear_congruential_engine.html

Errors and suggestions can be reported here: uray.janos@inf.elte.hu.