Last update: July 26, 2023 Translated From: e-maxx.ru

# Discrete Logarithm

The discrete logarithm is an integer $x$ satisfying the equation

$$a^x \equiv b \pmod{m}$$

for given integers $a$, $b$ and $m$.

The discrete logarithm does not always exist, for instance there is no solution to $2^x \equiv 3 \pmod{7}$. There is no simple condition to determine if the discrete logarithm exists.

In this article, we describe the **Baby-step giant-step** algorithm, an algorithm to compute the discrete logarithm proposed by Shanks in 1971, which has the time complexity $O(\sqrt{m})$. This is a **meet-in-the-middle** algorithm because it uses the technique of separating tasks in half.

## Algorithm

Consider the equation:

$$a^x \equiv b \pmod{m},$$

where $a$ and $m$ are relatively prime.

Let $x = np - q$, where $n$ is some pre-selected constant (we will describe how to select $n$ later). $p$ is known as **giant step**, since increasing it by one increases $x$ by $n$. Similarly, $q$ is known as **baby step**.

Obviously, any number $x$ in the interval $[0; m)$ can be represented in this form, where $p \in [1; \lceil \frac{m}{n} \rceil]$ and $q \in [0; n]$.

Then, the equation becomes:

$$a^{np-q} \equiv b \pmod{m}.$$

Using the fact that $a$ and $m$ are relatively prime, we obtain:

$$a^{np} \equiv ba^q \pmod{m}$$

This new equation can be rewritten in a simplified form:

$$f_1(p) = f_2(q).$$

This problem can be solved using the meet-in-the-middle method as follows:

- Calculate $f_1$ for all possible arguments $p$. Sort the array of value-argument pairs.
- For all possible arguments $q$, calculate $f_2$ and look for the corresponding $p$ in the sorted array using binary search.

## Complexity

We can calculate $f_1(p)$ in $O(\log m)$ using the binary exponentiation algorithm. Similarly for $f_2(q)$.

In the first step of the algorithm, we need to calculate $f_1$ for every possible argument $p$ and then sort the values. Thus, this step has complexity:

$$O\left(\left\lceil \frac{m}{n} \right\rceil \left(\log m + \log \left\lceil \frac{m}{n} \right\rceil\right)\right) = O\left(\left\lceil \frac{m}{n} \right\rceil \log m\right)$$

In the second step of the algorithm, we need to calculate $f_2(q)$ for every possible argument $q$ and then do a binary search on the array of values of $f_1$, thus this step has complexity:

$$O\left(n \left(\log m + \log \frac{m}{n}\right)\right) = O\left(n \log m\right).$$

Now, when we add these two complexities, we get $\log m$ multiplied by the sum of $n$ and $m/n$, which is minimal when $n = m/n$, which means, to achieve optimal performance, $n$ should be chosen such that:

$$n = \sqrt{m}.$$

Then, the complexity of the algorithm becomes:

$$O(\sqrt{m} \log m).$$

## Implementation

### The simplest implementation

In the following code, the function `powmod` calculates $a^b \pmod m$ and the function `solve` produces a proper solution to the problem. It returns $-1$ if there is no solution and returns one of

the possible solutions otherwise.

```
int powmod(int a, int b, int m) {
    int res = 1;
    while (b > 0) {
        if (b & 1) {
            res = (res * 1ll * a) % m;
        }
        a = (a * 1ll * a) % m;
        b >>= 1;
    }
    return res;
}

int solve(int a, int b, int m) {
    a %= m, b %= m;
    int n = sqrt(m) + 1;
    map<int, int> vals;
    for (int p = 1; p <= n; ++p)
        vals[powmod(a, p * n, m)] = p;
    for (int q = 0; q <= n; ++q) {
        int cur = (powmod(a, q, m) * 1ll * b) % m;
        if (vals.count(cur)) {
            int ans = vals[cur] * n - q;
            return ans;
        }
    }
    return -1;
}
```

In this code, we used `map` from the C++ standard library to store the values of $f_1$. Internally, `map` uses a red-black tree to store values. Thus this code is a little bit slower than if we had used an array and binary searched, but is much easier to write.

Notice that our code assumes $0^0 = 1$, i.e. the code will compute $0$ as solution for the equation $0^x \equiv 1 \pmod{m}$ and also as solution for $0^x \equiv 0 \pmod{1}$. This is an often used convention in algebra, but it's also not universally accepted in all areas. Sometimes $0^0$ is simply undefined. If you don't like our convention, then you need to handle the case $a = 0$ separately:

```
    if (a == 0)
        return b == 0 ? 1 : -1;
```

Another thing to note is that, if there are multiple arguments $p$ that map to the same value of $f_1$, we only store one such argument. This works in this case because we only want to return one possible solution. If we need to return all possible solutions, we need to change `map<int, int>` to, say, `map<int, vector<int>>`. We also need to change the second step accordingly.

## Improved implementation

A possible improvement is to get rid of binary exponentiation. This can be done by keeping a variable that is multiplied by $a$ each time we increase $q$ and a variable that is multiplied by $a^n$ each time we increase $p$. With this change, the complexity of the algorithm is still the same, but now the

$\log$ factor is only for the `map`. Instead of a `map`, we can also use a hash table (`unordered_map` in C++) which has the average time complexity $O(1)$ for inserting and searching.

Problems often ask for the minimum $x$ which satisfies the solution.
It is possible to get all answers and take the minimum, or reduce the first found answer using Euler's theorem, but we can be smart about the order in which we calculate values and ensure the first answer we find is the minimum.

```
// Returns minimum x for which a ^ x % m = b % m, a and m are coprime.
int solve(int a, int b, int m) {
    a %= m, b %= m;
    int n = sqrt(m) + 1;

    int an = 1;
    for (int i = 0; i < n; ++i)
        an = (an * 1ll * a) % m;

    unordered_map<int, int> vals;
    for (int q = 0, cur = b; q <= n; ++q) {
        vals[cur] = q;
        cur = (cur * 1ll * a) % m;
    }

    for (int p = 1, cur = 1; p <= n; ++p) {
        cur = (cur * 1ll * an) % m;
        if (vals.count(cur)) {
            int ans = n * p - vals[cur];
            return ans;
        }
    }
    return -1;
}
```

The complexity is $O(\sqrt{m})$ using `unordered_map`.

## When $a$ and $m$ are not coprime

Let $g = \gcd(a, m)$, and $g > 1$. Clearly $a^x \bmod m$ for every $x \geq 1$ will be divisible by $g$.

If $g \nmid b$, there is no solution for $x$.

If $g \mid b$, let $a = g\alpha, b = g\beta, m = g\nu$.

$$\begin{aligned} a^x &\equiv b \mod m \\ (g\alpha)a^{x-1} &\equiv g\beta \mod g\nu \\ \alpha a^{x-1} &\equiv \beta \mod \nu \end{aligned}$$

The baby-step giant-step algorithm can be easily extended to solve $ka^x \equiv b \pmod m$ for $x$.

```
// Returns minimum x for which a ^ x % m = b % m.
int solve(int a, int b, int m) {
```

```
    a %= m, b %= m;
    int k = 1, add = 0, g;
    while ((g = gcd(a, m)) > 1) {
        if (b == k)
            return add;
        if (b % g)
            return -1;
        b /= g, m /= g, ++add;
        k = (k * 1ll * a / g) % m;
    }

    int n = sqrt(m) + 1;
    int an = 1;
    for (int i = 0; i < n; ++i)
        an = (an * 1ll * a) % m;

    unordered_map<int, int> vals;
    for (int q = 0, cur = b; q <= n; ++q) {
        vals[cur] = q;
        cur = (cur * 1ll * a) % m;
    }

    for (int p = 1, cur = k; p <= n; ++p) {
        cur = (cur * 1ll * an) % m;
        if (vals.count(cur)) {
            int ans = n * p - vals[cur] + add;
            return ans;
        }
    }
    return -1;
}
```

The time complexity remains $O(\sqrt{m})$ as before since the initial reduction to coprime $a$ and $m$ is done in $O(\log^2 m)$.

## Practice Problems

- Spoj - Power Modulo Inverted
- Topcoder - SplittingFoxes3
- CodeChef - Inverse of a Function
- Hard Equation (assume that $0^0$ is undefined)
- CodeChef - Chef and Modular Sequence

## References

- Wikipedia - Baby-step giant-step
- Answer by Zander on Mathematics StackExchange

Contributors:
thanhtnguyen (35.24%) meooow25 (34.36%) wikku (15.86%) akashbhalotia (4.41%) jakobkogler (2.64%) adamant-pwn (2.64%) roll-no-1 (2.2%) likecs (1.32%) hieplpvip (0.88%) tcNickolas (0.44%)