Discrete Logarithm

Table of Contents

The discrete logarithm is an integer x solving the equation

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

where a and m are relatively prime. **Note**, if they are not relatively prime, then the algorithm described below is incorrect, though it can be modified so that it can work.

In this article, we describe the **Baby step - giant step** algorithm, proposed by Shanks in 1971, which has the time complexity $O(\sqrt{m} \log m)$. This algorithm is also known as **meet-in-the-middle** 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 exponentation 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. The resulting discrete logarithm can be big, but you can make it smaller using Euler's theorem.

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

int solve(int a, int b, int m) {
    int n = (int) sqrt (m + .0) + 1;
    map<int, int> vals;
    for (int p = n; p >= 1; --p)
        vals[powmod(a, p * n, m)] = p;
    for (int q = 0; q <= n; ++q) {
        int cur = (powmod(a, q, m) * 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.

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.

```
int solve(int a, int b, int m) {
    int n = (int) sqrt (m + .0) + 1;

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

    map<int, int> vals;
    for (int p = 1, cur = an; p <= n; ++p) {
        if (!vals.count(cur))
            vals[cur] = p;
        cur = (cur * an) % m;
    }

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

Practice Problems

- Spoj - Power Modulo Inverted
- Topcoder - SplittingFoxes3