CP-Algorithms

Page Authors

Search

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 incorret, 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 complexity $O(\sqrt{m} \log m)$. This algorithm is also known as **meet-in-the-middle**, because of it uses technique separation of 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 value of 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 rewriten in a simplified form:

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

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

- We calculate f_1 for all possible values of p. Sort these values.
- For each value of q, calculate f_2, and look for the corresponding value of p using the sorted array of f_1 using binary search.

Complexity

For each value of p, we can calculate $f_1(p)$ in $O(\log m)$ using binary exponentation algorithm. Similar for $f_2(q)$.

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

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

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

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

Now, when we add these two complexity, we would get $\log m$ multiplied by n and m/n, which has minimum value 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, function `powmod` performs binary exponential $a^b \pmod m$, and function `solve` produces a proper solution to the problem. It will returns -1 if there is no solution, and returns one possible solution in case a solution exists.

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

```
            else {
                a = (a * a) % m;
                b >>= 1;
            }
        return res % m;
    }

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

In this code, we used `map` from C++ STL to store the values of $f_1(i)$. Internally, `map` uses red-black-tree to store values. This code is a little bit slower than if we uses array and binary search for $f1$, but is much easier to write.

Another thing to note is that, if there are multiple values of p that has same value of f_1, we only store one such value. 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> >`. And we also need to change the second step accordingly.

Improved implementation

A possible improvement is to get rid of binary exponentiation in the second phase of the algorithm. This can be done by keeping a variable that multiplies by a each time we increase q. With this change, the complexity of the algorithm is still the same, but now the log part is only for map. Instead of map, we can also use hash table (`unordered_map` in GNU C++) which has complexity $O(1)$ for inserting and searching. And when the value of m is small enough, we can also get rid of map, and use a regular array to store and lookup values of f_1.

```
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 i=1, cur=an; i<=n; ++i) {
        if (!vals.count(cur))
            vals[cur] = i;
        cur = (cur * an) % m;
    }

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

Practice Problems

- Spoj - Power Modulo Inverted
- Topcoder - SplittingFoxes3