Discrete logarithm

Discrete logarithm problem is that according to an a, b, $m$solve the equation:

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

where $a$and m- **are relatively prime** (note: if they are not relatively prime, then the algorithm described below is incorrect, although presumably it can be modified so that it was still working).

Here we describe an algorithm, known as the **"baby-giant-STEP-STEP algorithm"** , proposed by **Shanks (Shanks)** in 1971, the running time for $O(\sqrt{m} \log m)$. Often simply referred to this algorithm algorithm **"meet-in-the-Middle"** (because it is one of the classic applications of technology "meet-in-the-middle": "separation of tasks in half").

Algorithm

So, we have the equation:

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

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

Transform equation. Put

$$x = np - q,$$

where n- is a preselected constant (as it is selected depending on m, we will understand later). Sometimes $p$called "giant step" (because the increase of its increases by one $x$at once n), as opposed to it q- "baby step".

Obviously, any x(the interval $[0; m)$- understandably, such that the range of values will suffice) can be represented in a form wherein it will be sufficient for the values:

$$p \in \left[1; \left\lceil \frac{m}{n} \right\rceil\right], \qquad q \in [0; n].$$

Then the equation becomes:

$$a^{np-q} = b \pmod{m},$$

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

$$a^{np} = ba^q \pmod{m}.$$

To solve the original equation, we need to find the appropriate values $p$and $q$to the values of the left and right parts of the match. In other words, it is necessary to solve the equation:

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

This problem is solved by the meet-in-the-middle as follows. The first phase of the algorithm: we can calculate the function $f_1$for all values of the argument p, and we can sort these values. The second phase of the algorithm: we sort out the value of the second variable q, compute the second function f_2, and look for the value of the predicted values of the first function using a binary search.

Asymptotics

First, we estimate the computation time of each of the functions $f_1(p)$and $f_2(q)$. And she and the other contains the exponentiation which can be done using the algorithm of binary exponentiation . Then both of these functions, we can compute in time $O(\log m)$.

The algorithm in the first phase comprises computing functions $f_1(p)$for each possible value $p$and further sorting of values that gives us the asymptotic behavior:

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

In the second phase of the algorithm is evaluated functions $f_2(q)$for each possible value $q$and a binary search on an array of values $f_1$that gives us the asymptotic behavior:

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

Now, when we combine these two asymptotic we get $\log m$multiplied by the sum $n$and m/n, and almost obvious that the minimum is reached when $n \approx m/n$, ie algorithm for optimal constant $n$should be chosen:

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

Then the asymptotic behavior of the algorithm takes the form:

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

Note. We could exchange roles f_1 and f_2 (ie the first phase to compute values of the function f_2, and the second - f_1), but it is easy to understand that the result will not change, and the asymptotic behavior of this we can not improve.

Implementation

The simplest implementation

The function powmod performs binary raising a number a to the power b mod m, see binary exponentiation .

Function solve produces the actual solution of the problem. This function returns a response (the number in the interval $[0; m)$), or more precisely, one of the answers. The function will return -1 if there is no solution.

```
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;
}
```

Here we are for the convenience of the implementation of the first phase of the algorithm used the data structure "map" (red-black tree) that for each value of the function $f_1(i)$stores the argument $i$in which this value is reached. Here, if the same value is achieved repeatedly recorded smallest of all the arguments. This is done in order to subsequently, on the second phase of the algorithm is found in the response interval $[0; m)$.

Given that the argument of $f_1()$the first phase we iterates from one and up n, and argument of $f_2()$the second phase moves from zero to n, then eventually we cover the whole set of possible answers, because segment $[0; n^2]$contains a gap $[0; m)$. At the same negative response could happen, and the responses, greater than or equal m, we can not ignore - should still be in the corresponding period of the answers $[0; m)$.

This function can be changed in the event if you want to find **all solutions** of the discrete logarithm. To do this, replace "map" on any other data structure that allows for a single argument to store multiple values (for example, "multimap"), and to amend the code of the second phase.

An improved

When **speed optimization** can proceed as follows.

First, immediately struck by the uselessness of the binary exponentiation algorithm in the second phase. Instead, you can just start a variable and multiplies it every time a.

Secondly, the same way you can get rid of the binary exponentiation, and the first phase: in fact, once is enough to calculate the magnitude $a^n$and then just multiplies it.

Thus, the logarithm in the asymptotic behavior will remain, but it will only logarithm associated with the data structure map <>(ie, in terms of the algorithm, with sorting and binary search values) - ie it will be the logarithm of $\sqrt{m}$that in practice provides a notable acceleration.

```
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;
}
```

Finally, if the unit $m$is small enough, then we can do to get rid of the logarithm in the asymptotic behavior - instead of just having got the map <>usual array.

You can also recall the hash table: on average, they work well for $O(1)$that, in general gives an asymptotic $O(\sqrt{m})$.