MAXimal

added: 10 Jun 2008 18:08
Edited: 23 Mar 2012 3:50

Discrete logarithm

Contents [hide]

The 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, though, presumably, it can be modified so that it is still working).

Here we describe an algorithm, known as the **"Baby-Step-giant-Step algorithm"** , proposed by **Shanks (Shanks)** in 1971, working at the time for $O(\sqrt{m} \log m)$. Often, this algorithm is simply called an algorithm **"meet-in-the-middle"** (because it is one of the classic applications 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 her chosen depending on m, we will understand later). Sometimes $p$called "giant step" (because increasing it by one increases $x$at once n), as opposed to it q- "baby step".

It is clear that any x(in the interval $[0; m)$- it is clear that such a range of values will be enough) can be represented in a form where, for this will be enough values:

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

Then the equation becomes:

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

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

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

In order to solve the original equation, you 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 using the meet-in-the-middle as follows. The first phase of the algorithm: calculate the value of the function $f_1$for all values of the argument p, and we can sort the values. The second phase of the algorithm: Let's take 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 construction of the power that can be performed using the algorithm binary exponentiation . Then both of these functions, we can compute in time $O(\log m)$.

The algorithm itself in the first phase comprises computing functions $f_1(p)$for each possible value $p$and the 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 function $f_2(q)$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 add these two asymptotes, we can do it $\log m$, multiplied by the sum $n$and m/n, and almost obvious that the minimum is attained when $n \approx m/n$, that is, for optimum performance of the algorithm 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 calculate the 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 that we can not improve.

Implementation

The simplest implementation

The function powmodperforms a binary construction of $a$the power $b$mod m, see. binary exponentiation .

The function produces a proper solution to the problem. This function returns a response (the number in the interval $[0; m)$), or more precisely, one of the answers. Function will return -1if 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 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 to ensure that subsequently, in the second phase of the algorithm found in the response interval $[0; m)$.

Given that the argument of the first phase we pawing away from one and up n, and the argument of the function in the second phase moves from zero to n, in the end, we cover the entire set of possible answers, because segment contains a gap $[0; m)$. In this case, the negative response could not get, and the responses of 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 that if you want to find **all the solutions** of the discrete logarithm. To do this, replace the "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 **optimizing for speed** , you can proceed as follows.

Firstly, immediately catches the eye uselessness binary exponentiation in the second phase of the algorithm. Instead, you can just make it multiply the variable and every time a.

Secondly, in the same way you can get rid of the binary exponentiation, and in the first phase: in fact, once is enough to calculate the value , and then simply multiply the at her.

Thus, the logarithm of the asymptotic behavior will remain, but it will only log associated with the data structure (ie, in terms of the algorithm, with sorting and binary search values) - ie it will be the logarithm of that in practice gives a noticeable boost.

```
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, it can and does get rid of the logarithm in the asymptotic behavior - instead of just having got a regular array.

You can also recall the hash table: on average, they work well for that, in general, gives the asymptotic behavior .

6 Комментариев e-maxx Войти

Лучшее вначале Поделиться Избранный

Присоединиться к обсуждению...

Евгений • 2 года назад
Отличная статья, спасибо!
4 • Ответить • Поделиться ›

Kartikeya Bhardwaj • год назад
I wish it was in english. .google translator sucks. .still rele nice. .thnx e-maxx
1 • Ответить • Поделиться ›

Roman Dryndik → Kartikeya Bhardwaj • 4 месяца назад
Do you still need it in english?
• Ответить • Поделиться ›

Guest • 2 года назад
Really amazing explanation :) Proof: I just solved a problem using this algorithm
http://www.lightoj.com/volume_...
1 • Ответить • Поделиться ›

Igorigorii • 2 года назад
Hello
1 • Ответить • Поделиться ›

Mohammad Samiul Islam • 2 года назад
Really nice explanation :) Proof: I just solved a problem after learning the algorithm here :D
http://www.lightoj.com/volume_...
• Ответить • Поделиться ›