Last update: June 6, 2022 Translated From: e-maxx.ru

Primitive Root

Definition

In modular arithmetic, a number g is called a `primitive root modulo n` if every number coprime to n is congruent to a power of g modulo n. Mathematically, g is a `primitive root modulo n` if and only if for any integer a such that $\gcd(a, n) = 1$, there exists an integer k such that:

$g^k \equiv a \pmod n.$

k is then called the `index` or `discrete logarithm` of a to the base g modulo n. g is also called the `generator` of the multiplicative group of integers modulo n.

In particular, for the case where n is a prime, the powers of primitive root runs through all numbers from 1 to $n-1$.

Existence

Primitive root modulo n exists if and only if:

- n is 1, 2, 4, or
- n is power of an odd prime number $(n = p^k)$, or
- n is twice power of an odd prime number $(n = 2 \cdot p^k)$.

This theorem was proved by Gauss in 1801.

Relation with the Euler function

Let g be a primitive root modulo n. Then we can show that the smallest number k for which $g^k \equiv 1 \pmod n$ is equal $\phi(n)$. Moreover, the reverse is also true, and this fact will be used in this article to find a primitive root.

Furthermore, the number of primitive roots modulo n, if there are any, is equal to $\phi(\phi(n))$.

Algorithm for finding a primitive root

A naive algorithm is to consider all numbers in range $[1, n-1]$. And then check if each one is a primitive root, by calculating all its power to see if they are all different. This algorithm has complexity $O(g \cdot n)$, which would be too slow. In this section, we propose a faster algorithm using several well-known theorems.

From previous section, we know that if the smallest number k for which $g^k \equiv 1 \pmod n$ is $\phi(n)$, then g is a primitive root. Since for any number a relative prime to n, we know from Euler's theorem that $a^{\phi(n)} \equiv 1 \pmod n$, then to check if g is primitive root, it is enough to check that for all d less than $\phi(n)$, $g^d \not\equiv 1 \pmod n$. However, this algorithm is still too slow.

From Lagrange's theorem, we know that the index of 1 of any number modulo n must be a divisor of $\phi(n)$. Thus, it is sufficient to verify for all proper divisor $d \mid \phi(n)$ that $g^d \not\equiv 1 \pmod n$. This is already a much faster algorithm, but we can still do better.

Factorize $\phi(n) = p_1^{a_1} \cdots p_s^{a_s}$. We prove that in the previous algorithm, it is sufficient to consider only the values of d which have the form $\frac{\phi(n)}{p_j}$. Indeed, let d be any proper divisor of $\phi(n)$. Then, obviously, there exists such j that $d \mid \frac{\phi(n)}{p_j}$, i.e. $d \cdot k = \frac{\phi(n)}{p_j}$. However, if $g^d \equiv 1 \pmod n$, we would get:

$$g^{\frac{\phi(n)}{p_j}} \equiv g^{d \cdot k} \equiv (g^d)^k \equiv 1^k \equiv 1 \pmod n.$$

i.e. among the numbers of the form $\frac{\phi(n)}{p_i}$, there would be at least one such that the conditions were not met.

Now we have a complete algorithm for finding the primitive root:

- First, find $\phi(n)$ and factorize it.
- Then iterate through all numbers $g \in [1, n]$, and for each number, to check if it is primitive root, we do the following:
  - Calculate all $g^{\frac{\phi(n)}{p_i}} \pmod n$.
  - If all the calculated values are different from 1, then g is a primitive root.

  Running time of this algorithm is $O(Ans \cdot \log \phi(n) \cdot \log n)$ (assume that $\phi(n)$ has $\log \phi(n)$ divisors).

Shoup (1990, 1992) proved, assuming the generalized Riemann hypothesis, that g is $O(\log^6 p)$.

Implementation

The following code assumes that the modulo `p` is a prime number. To make it works for any value of `p`, we must add calculation of $\phi(p)$.

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

int generator (int p) {
    vector<int> fact;
    int phi = p-1,  n = phi;
    for (int i=2; i*i<=n; ++i)
        if (n % i == 0) {
            fact.push_back (i);
            while (n % i == 0)
                n /= i;
        }
    if (n > 1)
        fact.push_back (n);

    for (int res=2; res<=p; ++res) {
        bool ok = true;
        for (size_t i=0; i<fact.size() && ok; ++i)
            ok &= powmod (res, phi / fact[i], p) != 1;
        if (ok)  return res;
    }
    return -1;
}
```

Contributors:
thanhtnguyen (81.25%) hoke-t (9.38%) adamant-pwn (6.25%) algmyr (2.08%) jakobkogler (1.04%)

primitive root of a very big prime number (Elgamal DS)

Asked 5 years, 3 months ago Modified 5 years, 3 months ago Viewed 2k times

1

For encryption methods there is a need for (very large) prime numbers. So for Elgamal digital signature there is the problem of finding a primitive root of (p) where p must be very large prime number (1024) or (2048) bits length.

My question is: how does the encrypter solve this problem?

I presume a loop to check all the possibilities is an option.

encryption signature prime-numbers elgamal-signature

Share Improve this question Follow

edited Mar 4, 2018 at 22:53
Maart Maarten Bodewes - on strike ♦
90.3k 13 158 309

asked Mar 4, 2018 at 19:06
Q.M.J
21 2

1 Dupe crypto.stackexchange.com/questions/54254/... although the ossifrageal (?) answer is less tutorial than the ursine one. – dave_thompson_085 Mar 5, 2018 at 6:25

1 Answer

Sorted by: Highest score (default)

8

When considering a big prime p, the group of invertible integers modulo p are all integers from 1 to $p-1$. There are $p-1$ of them. The *order* of an integer g modulo p is the smallest integer $k > 0$ such that $g^k = 1 \pmod p$. Group theory states that the powers of an element g, i.e. 1, g, g^2... are collectively a subgroup of the group of invertible integers modulo p, and the order of g is really the size of that subgroup (called "the subgroup generated by g"). The cardinal of a subgroup is always a divisor of the cardinal of the group that contains it. Thus, k divides $p-1$.

A *primitive* element g is one such that the subgroup it generates really is all of the invertible integers modulo p, not just some of them. Therefore, to verify whether an integer g is primitive or not, all you have to do is check that its order is $p-1$ but not one of the other possible subgroup orders. That is, you check that $g^{k'} \neq 1 \pmod p$ for all $k' < p-1$ such that k' divides $p-1$.

The problem can be simplified as follows: suppose that there is a $k' < p-1$, such that k' divides $p-1$, and $g^{k'} = 1 \pmod p$. We can consider $\alpha = (p-1)/k'$, which is an integer greater than 1, and thus a product of some prime integers (since every integer can be decomposed as a product of prime factors). Let's call r one of the prime factors of α (i.e. we write $\alpha = r\beta$ for some integer β). Since α divides $p-1$, so does r, and we can compute:

$$k'' = (p-1)/r = \beta k'$$

Thus:

$$g^{k''} = (g^{k'})^{\beta} = 1^{\beta} = 1 \pmod{p}$$

In other words, if g is *not* primitive, then there must be a prime integer r that divides $p-1$ such that $g^{(p-1)/r} = 1 \pmod{p}$.

This means that in order to test whether a given g is primitive, then it suffices to apply the following test:

> Let p be a prime, and g a non-zero integer modulo p. If, for all primes r that divide $p-1$, $g^{(p-1)/r} \neq 1 \pmod{p}$, then g is primitive, i.e. its order modulo p is exactly $p-1$.

Now comes the actual difficulty: you must somehow know the list of prime factors of $p-1$. This is easy if you generate p "specially", as a so-called "safe prime" (the terminology "safe" here is traditional and does not actually imply that the prime is inherently "safer" than any other, but it makes our problem easier). A "safe prime" is an integer p which is such that both p and $(p-1)/2$ are prime. If you generate your modulus for ElGamal signatures are a "safe prime", then there are only two prime factors of $p-1$: these are 2, and $(p-1)/2$. The test above now becomes the following:

- Get a random non-zero g modulo p.
- Check that $g^2 \neq 1 \pmod{p}$. Note that there are only two integers g modulo p such that $g^2 = 1 \pmod{p}$, and these are 1 and $p-1$. Thus, it suffices to choose g greater than 1 and lower than $p-1$ to fulfill that condition.
- Check that $g^{(p-1)/2} \neq 1 \pmod{p}$. If that condition is not met, then you must start over with another g. Otherwise, that's it, you have a primitive g.

No specific value of g is better than any other with regards to resistance to discrete logarithm, so you might just as well try small values of g, that offer a small performance boost; you start with $g = 2$ and simply increment it until you reach a primitive value. Exactly half of the integers from 2 to $p-2$ are primitives, so you do not have to try many times before reaching such a value.

IMPORTANT: Take care that the test above is for a "safe prime". In the general case, if given a prime p without any knowledge of how it was generated, you would have to *factorize* $p-1$ into its prime factors, a problem which is, in all generality, quite hard to solve for large integers.

While the ElGamal scheme is nominally defined with a primitive root that generates all invertible integers modulo p, it can also be applied to a subgroup, at which point it becomes mostly the same thing as DSA. In DSA, it is customary to generate p as follows:

- Let q be a random medium-sized prime (e.g. 256 bits).
- Let p be a random large prime (e.g. 2048 bits) such that q divides $p-1$. In practice, this means that random integers z are produced, and, for each, p is computed as $p = qz + 1$, until a prime p is obtained.
- To make an element g of order exactly q, a random integer m modulo p is generated, and we set $g = m^{(p-1)/q} \bmod p$. It is easily shown that $g^q = 1 \pmod p$, which means that the order of g is a divisor of q. Since q has been chosen to be prime, the order of g can be either 1 or q. If $g \neq 1$, then its order cannot be 1. Therefore, g will have order exactly q as long as it is not equal to 1. Probability of g being equal to 1 is very very small, but even if that happened, then you would just have to start again with another value m.

Summary: to get a primitive element modulo p, you get random values and then *check* whether you obtained a primitive element. Primitive elements are not rare, so this works well in practice. However, to do the check, you have to know the factorization of $p-1$, and that is difficult in the general case. Thus, the usual method is to produce the modulus p with a generation method that also lets you know the factorization of $p-1$, e.g. you make p as a "safe prime". Alternatively, you do not look for a primitive element, but for the generator of a subgroup of known prime size (as is done in DSA).