CP-Algorithms

Page Authors

Search

Primitive Root

Table of Contents

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 $(n = 2. p^k)$.

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 $O(g.n)$, 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 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 $\phi(n) = p_1^{a_1} \dots p_s^{a_s}$. We prove that in the previous algorithm, it is sufficient to consider only the values of d which has 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. $d.k = \frac{\phi(n)}{p_j}$. However, if $g^d \equiv 1 \pmod n$, we would get:

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

i.e. among the numbers of the form $\frac{\phi(n)}{p_i}$, there would be at least one such that the conditions are 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 $g = 1 \ldots n$, 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 $O(Ans.log\phi(n).\log n)$ (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),  --
        else
            a = int (a * 1ll * a % p),  b >>=
    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;
            ok &= powmod (res, phi / fact[i],
        if (ok)  return res;
    }
    return -1;
}
```