Math: tips, tricks and hacks

Vladimir Smykalov

Toulouse, 2017

Finding modular inverse

Problem description

Input: `a`, `m`, so that `gcd(a , m) = 1`

Output: `x`, so that `a * x == 1 (mod m)`

Prime module

Fermat's little theorem: `a^(p-1) == 1 (mod p)`

So, `a * a^(p-2) == 1 (mod p)`

Then `x = a^(p-2)` is good

Just use binary exponentiation

General case (not-prime modules)

Solution 1: Euler's theorem

Solution 2: Euclidean algorithm

Solution 3: magic

General case (not-prime modules)

Solution 1: Euler's theorem

Euler's theorem: `a ^ phi(m) == 1 (mod m)`
Where `phi(m)` -- the number of `1 <= x <= m` such that `gcd(x, m) = 1`

Find `phi(m)` than use binary exponentiation

Quick note on finding `phi(m)`

That's easy!

According to math, if `m = p1^k1 * p2^k2 * ...`
than `phi(m) = m * (p1-1)/p1 * (p2-1)/p2 * ...`

So, can be easily found in $O(\sqrt{m})$

```
int get_phi(int m)
{
    int r = m;
    for (int i = 2; i * i <= m; ++i) if (m % i == 0)
    {
        r = r / i * (i - 1);
        while (m % i == 0) m /= i;
    }
    if (m > 1) r = r / m * (m - 1);
    return r;
}
```

General case (not-prime modules)

Solution 2: Euclidean algorithm

Since `gcd(a, m) = 1`, there **exist** `x` and `y`,
such that `a * x + m * y = 1`

Then `a * x == 1 (mod m)`

General case (not-prime modules)

Solution 3: magic

```
int rev(int a, int m)
{
    if (a == 1) return 1;
    return (1 - rev(m % a, a) * (long long)m) / a + m;
}
```

Chinese Reminder Theorem

Problem description

Input: pairs `(a1, m1), (a2, m2), ...`, such that `gcd(m_i, m_j) = 1` for every `i`, `j`
Output: `x` such that `x == a_i (mod m_i)` for every `i`

Chinese Reminder Theorem

Lets solve this for two pairs `(a1, m1)` and `(a2, m2)`
(after that we will be able to combine them into one `(x, m1 * m2)` and move on to next pairs)

Want `x == a1 (mod m1)` and `x == a2 (mod m2)`

Let `x = a1 + k * m1`

Want `a1 + k * m1 == a2 (mod m2)`

`k * m1 == a2 - a1 (mod m2)`

`k == (a2 - a1) * rev(m1, m2) (mod m2)`

Profit!

Primitive roots

Let `p` be prime number

Then there exist `g` such that:

$g^0, g^1, g^2, \ldots, g^{(p-2)}$ is a permutation of `1, 2, ..., p-1`

Primitive roots

How to check is `g` is primitive root

If `g` is primitive root, then the smallest `a` such that `g^a == 1 (mod p)` is `a = p-1`
Also, for every smallest such `a` we have `(p-1) % a = 0`

So... if for **every** `x` such that `(p-1) % x = 0` we have `g^x != 1 (mod p)` than `g` is primitive root

Primitive roots

How to find primitive root

Fact: smallest primitive root is **very-very small**

So, we can iterate `g` from `1` to `(p-1)` and `check` every time. Once `check` returns `true`, we found it

Rumors say that the smallest primitive root is $O(\log \log p)$

Discrete logarithm

Problem description

Input: `g`, `a` and prime module `p`
Output: `x` such that `g^x == a (mod p)`

Discrete logarithm

Algo is called **baby-step giant-step**

Suppose `m` is approx. `sqrt(p)`, but is strictly greater than `sqrt(p)`

Now, let `x = y * m - z`, where `1 <= y <= m` and `1 <= z <= m`
(this scheme allows us every number from `0` to `m^2 - 1`)

Want `g^x == a`, so `g ^ (y*m-z) == a`
`g^(y*m) == a * g^z (mod p)`

Now we have `m` possible values for left side of equation (for each `y`)
Same way we have `m` possible values for right side (for each `z`)

So we can check if there is a match in $O(m \log m)$
So... the total run-time of algo is $O(\sqrt{p} \log p)$

Calculating combinations `(n, k) % mod` for `n <= 10^6`

Sometimes math problems forces us to quickly compute $\binom{n}{k}$ in $O(1)$ time

Way 1: (basic) precalculate `(n, k)` for all possible pairs in $O(n^2)$ time, using relation `(n, k) = (n-1, k-1) + (n-1, k)`. But that's too slow for large `n`

Calculating combinations `(n, k) % mod` for `n <= 10^6`

Way 2: precalculate `fact(n)` and `rev(fact(n))` for each `n`.

After that with relation $\binom{n}{k} = \frac{n!}{k!(n-k)!}$ we can compute in $O(1)$ time

But how to precalculate `rev(fact(n))` in `O(n)` time?

Step 1: Calculate `fact(n)` for the largest `n` in $O(n)$

Step 2: Calculate `rev(fact(n))` for largest `n` in $O(\log)$

Step 3: Use `rev(fact(k)) = rev(fact(k+1)) * (k+1)` to evaluate values for smaller `n` in $O(n)$

So in total we can after precalc in $O(n + \log)$ time we can compute $\binom{n}{k}$ in $O(1)$ time