Discrete Logarithm

Logarit rời rạc

Người viết:

- Nguyễn Minh Hiển - Trường Đại học Công nghệ, ĐHQGHN

Reviewer:

- Đặng Đoàn Đức Trung - UT Austin
- Phạm Công Minh - Trường Đại học Công nghệ - ĐHQGHN

Giới thiệu

- Logarit rời rạc được định nghĩa là số nguyên x thỏa mãn phương trình: $a^x \equiv b \pmod m$ với a, b, m cho trước
- Logarit rời rạc không phải lúc nào cũng tồn tại. Ví dụ: không tồn tại x sao cho: $2^x \equiv 3 \pmod 7$ và cũng không có điều kiện cụ thể để xác định xem x tồn tại không
- Trong bài viết này, chúng ta sẽ tìm hiểu thuật toán "Baby-step giant-step" được đề xuất bởi Shanks năm 1971 với độ phức tạp thời gian $\mathcal{O}(\sqrt{m})$.

Thuật toán Baby step - Giant Step

Với $\gcd(a, m) = 1$, xét phương trình:

$$a^x \equiv b \pmod m$$

Ta cần tìm $x \in [0; m)$

- **Biến đổi bài toán:** Ta đặt $x = np - q$ với n là một hằng số được chọn trước (ta sẽ có cách chọn n sau). Khi này p được gọi là *"giant-step" (bước lớn)* vì tăng nó 1 đơn vị sẽ tăng x thêm n. Tương tự thì q được gọi là *"baby-step" (bước nhỏ)*.
  Rõ ràng, bất kỳ số nào trong đoạn $[0; m)$ đều biểu diễn được dưới dạng này, với $p \in \left[1; \left\lceil \frac{m}{n} \right\rceil\right]$ và $q \in [0; n]$
  Lúc này phương trình trở thành:

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

Vì $\gcd(a, m) = 1$, ta có:

$$a^{np} \equiv ba^q \pmod m$$

Đặt $f_1(p) = a^{np}$ và $f_2(q) = a^q$, phương trình trở thành:

$f_1(p) = f_2(q)$

Nếu tìm được p, q thỏa mãn, ta sẽ tìm ra nghiệm $x = np - q$.

Nếu bạn đọc chỉ cần tìm nghiệm x dương nhỏ nhất, hãy chú ý nhận xét sau:

> **Nhận xét:** Nếu $p_1 > p_2$ thì $np_1 - q_1 > np_2 - q_2$.
> Điều này có thể chứng minh dễ dàng bằng phản chứng.

Từ nhận xét này, ta có thuật toán sau bao gồm 3 bước:

- **Bước 1:**
  - Duyệt q từ 0 đến n:
    - Tính và lưu giá trị $f_2(q) = b \cdot a^q \mod m$ vào một mảng.
  - Độ phức tạp: $\mathcal{O}(n)$
- **Bước 2:**
  - Sắp xếp các giá trị của $f_2(q)$ (*để thực hiện tìm kiếm nhị phân ở bước tiếp theo*).
  - Độ phức tạp: $\mathcal{O}(n \log n)$
- **Bước 3:**
  - Đầu tiên, ta sử dụng lũy thừa nhanh tính $a^n \mod m$.
  - Sau đó, duyệt p từ 1 đến $\left\lceil \frac{m}{n} \right\rceil$:
    - Tính $f_1(p) = (a^n)^p \mod m$.
    - Kiểm tra xem có tồn tại q sao cho $f_1(p) = f_2(q)$ không bằng *thuật toán tìm kiếm nhị phân*. Nếu tồn tại, $x = np - q$ là kết quả cần tìm.
  - Độ phức tạp: $\mathcal{O}\left(\left\lceil \frac{m}{n} \right\rceil \log n\right)$

Tổng kết lại, độ phức tạp thuật toán là: $\mathcal{O}\left(n \log n + \left\lceil \frac{m}{n} \right\rceil \log n\right)$.

Chọn $n = \sqrt{m}$, ta được độ phức tạp: $\mathcal{O}\left(\sqrt{m} \log m\right)$.

Code C++ minh họa:

```
int binpow(long long a, long long k, int mod){
    long long res = 1;
    for (; k; k >>= 1, a = a * a % mod){
        if (k & 1)
            res = res * a % mod;
    }
    return res;
}

int discrete_log_BSGS_naive(int a, int b, int m) {
    a %= m;
```

```
    b %= m;
    int n = sqrt(m) + 1;

    vector<pair<int, int>> f2(n + 1);
    for (int q = 0, cur = b; q <= n; q++) {
        f2[q] = make_pair(cur, q);
        cur = 1LL * cur * a % m;
    }
    sort(f2.begin(), f2.end());

    int step = binpow(a, n, m);
    for (int p = 1, f1 = 1; p <= n; p++) {
        f1 = 1LL * f1 * step % m;
        auto res = *lower_bound(f2.begin(), f2.end(), make_pair
        if (res.first == f1) {
            return n * p - res.second;
        }
    }
    return -1;
}
```

Cải tiến với Bảng băm

$a^x \equiv b \pmod m$

Để cải tiến thuật toán, ta sử dụng cấu trúc dữ liệu bảng băm (Hash Table).
Với C++, ta có thể sử dụng `std::unordered_map`.

Xét hàm băm $\mathrm{vals} : \mathbb{N} \to \mathbb{N}$ thỏa mãn:

$$\mathrm{vals}(x) \text{ có giá trị là } q \implies f_2(q) = x.$$
$$\text{Nếu không tồn tại } q \text{ để } f_2(q) = x \text{ thì } \mathrm{vals}(x) \text{ không được gán giá trị nào.}$$

- **Bước 1:**
  - Duyệt q từ 0 đến n:
    - Tính $f_2(q) = b \cdot a^q \mod m$ và lưu lại $\mathrm{vals}\big(f_2(q)\big) = q$.
  - Độ phức tạp là $\mathcal{O}(n)$.
- **Bước 2:** Tương tự bước 3 ở trên
  - Đầu tiên, ta sử dụng lũy thừa nhanh tính $a^n \mod m$.
  - Sau đó, duyệt p từ 1 đến $\lceil \frac{m}{n} \rceil$:
    - Tính $f_1(p) = (a^n)^p \mod m$.
    - *Sử dụng hàm băm vals thay cho tìm kiếm nhị phân*:
      Nếu $\mathrm{vals}\big(f_1(p)\big)$ có giá trị là q thì $x = np - q$ là kết quả cần tìm.
  - Độ phức tạp là $\mathcal{O}\left(\lceil \frac{m}{n} \rceil\right)$.

Tổng kết lại, độ phức tạp bài toán là: $\mathcal{O}\left(n + \lceil \frac{m}{n} \rceil\right)$.

Tương tự, chọn $n = \sqrt{m}$, ta được độ phức tạp: $\mathcal{O}\left(\sqrt{m}\right)$.

Code C++ minh họa:

Trong C++, ta sử dụng `unordered_map` như một bảng băm:

- `vals.count(x) > 0` sẽ kiểm tra xem có tồn tại $\text{vals}[x]$ không
- `vals[x]` có là giá trị q thỏa mãn $f_2(q) = x$

```cpp
int discrete_log_BSGS_coprime(int a, int b, int m) {
    a %= m, b %= m;
    int n = sqrt(m) + 1;

    unordered_map<int, int> vals;
    for (int q = 0, cur = b; q <= n; ++q) {
        vals[cur] = q;
        cur = 1LL * cur * a % m;
    }

    int step = binpow(a, n, m);
    for (int p = 1, f1 = 1; p <= n; p++) {
        f1 = 1LL * f1 * step % m;
        if (vals.count(f1)) {
            return n * p - vals[f1];
        }
    }
    return -1;
}
```

Trường hợp gcd(a,m) khác 1

- Đặt $g = \gcd(a, m) > 1$. Rõ ràng $a^x \mod m$ chia hết cho g với mọi $x \geq 1$.
- Nếu $g \nmid b$, không có nghiệm x.
- Nếu $g \mid b$, đặt $a = g\alpha,\ b = g\beta,\ m = g\nu$

$$\begin{aligned} a^x &\equiv b \mod m \\ (g\alpha)a^{x-1} &\equiv g\beta \mod g\nu \\ \alpha a^{x-1} &\equiv \beta \mod \nu \end{aligned}$$

- Thuật toán "baby-step giant-step" có thể dễ dàng giải được $ka^x \equiv b \pmod m$

Code C++ minh họa:

```cpp
// Trả về x nhỏ nhất thỏa mãn a ^ x % m = b % m.
int discrete_log_BSGS(int a, int b, int m) {
    a %= m, b %= m;
    int n = sqrt(m) + 1;

```

```
    int k = 1, add = 0, g;
    while ((g = __gcd(a, m)) > 1) {
        if (b == k) return add;
        if (b % g) return -1;
        b /= g, m /= g, ++add;
        a %= m;
        k = (k * 1ll * a / g) % m;
    }

    unordered_map<int, int> vals;
    for (int q = 0, cur = b; q <= n; ++q) {
        vals[cur] = q;
        cur = 1LL * cur * a % m;
    }

    int step = binpow(a, n, m);
    for (int p = 1, f1 = k; p <= n; p++) {
        f1 = 1LL * f1 * step % m;
        if (vals.count(f1)) {
            int ans = n * p - vals[f1] + add;
            return ans;
        }
    }
    return -1;
}
```

Như bạn đọc thấy, thuật toán vẫn khá giống **Cải tiến với bảng băm**. Để đưa về trường hợp $\gcd(a, m) = 1$, ta chỉ tốn thêm bước xử lý sau:

```
while ((g = __gcd(a, m)) > 1) {
        ...
        m /= g;
        ...
}
```

Mỗi vòng ta chia m cho một số $g > 1$, nên chỉ có tối đa $\log_2 m$ vòng. Mà mỗi vòng ta tính $\gcd(a, m)$, nên tổng ĐPT cho phần xử lý này xấu nhất là $\log^2 m$.

Tổng kết lại, độ phức tạp thuật toán vẫn là $\mathcal{O}\left(\sqrt{m}\right)$.

Cấp của số nguyên

Định nghĩa

Cho $\gcd(a, m) = 1$. Gọi h là số nguyên dương *nhỏ nhất* thỏa mãn

$$a^h \equiv 1 \pmod m$$

Khi này h được gọi là **cấp của số nguyên** a **modulo** m. Ký hiệu: $h = \text{ord}_m(a)$

Tính chất

- Nếu $a^k \equiv 1 \pmod m$ thì $\text{ord}_m(a) \mid k$.
- Vì $a^{\varphi(m)} \equiv 1 \pmod m$ với $\varphi(m)$ là hàm phi Euler nên:

$$\text{ord}_m(a) \mid \varphi(m)$$

Thuật toán tìm cấp số nguyên

Bài toán: Cho $\gcd(a, m) = 1$. Tìm $\text{ord}_m(a)$ hay số nguyên dương x *nhỏ nhất* thỏa mãn

$$a^x \equiv 1 \pmod m$$

Tương tự như trên, ta có thể sử dụng thuật toán *Baby step - Giant step* bên trên. Tuy nhiên với 2 tính chất nêu trên của $\text{ord}_m(a)$, ta có thuật toán tốt hơn dưới đây:

- **Bước 1:** Tính và phân tích thừa số nguyên tố của $\varphi(m) = p_1\ p_2 \dots p_r$ trong đó p_i nguyên tố và không cần phân biệt.
  $\longrightarrow$ Thuật toán phân tích thừa số nguyên tố Pollard's rho sẽ giúp bước này có ĐPT tổng thể là $\mathcal{O}\left(\sqrt[4]{m} \log m\right)$.
- **Bước 2:** Gán $\text{res} := \varphi(m)$.
  Duyệt i từ 1 đến r:
  Nếu $a^{\text{res}/p_i} \equiv 1 \pmod m$ thì gán $\text{res} := \text{res}/p_i$.
  $\longrightarrow$ Do $r \le \log\left(\varphi(m)\right) \le \log m$ nên ĐPT bước này là $\mathcal{O}(\log^2 m)$.

Sau cùng, res chính là $\text{ord}_m(a)$ cần tìm.
Tổng độ phức tạp thuật toán trên là $\mathcal{O}\left(\sqrt[4]{m} \log m\right)$.

Code C++ minh họa:

```cpp
using ll = long long;
ll powMod(ll x, ll p, ll md);
ll gcd(ll x, ll y);
// danh sách các ước nguyên tố của x (có thể trùng nhau)
vector<ll> factorize(ll x);
// hàm phi Euler
ll phi(ll n) {
    auto ps = factorize(n);
    ll res = n;
    ll last = -1;
    for (auto p : ps) {
        if (p != last) {
            res = res / p * (p - 1);
            last = p;
        }
    }
    return res;
}
// Cấp của a mod m
ll ord(ll a, ll m) {
    if (gcd(a, m) != 1) return -1;
```

```
    ll res = phi(m);
    auto ps = factorize(res);
    for (auto p : ps)
        if (powMod(a, res / p, m) == 1) res /= p;
    return res;
}
```

▶ *Code C++ đầy đủ*

Tổng kết

- Thuật toán *Baby step - Giant step*, kết hợp sử dụng bảng băm và xử lý trường hợp $\gcd(a, m) \neq 1$ mất ĐPT $\mathcal{O}\left(\sqrt{m}\right)$.
- Trong trường hợp đặc biệt là $a^x \equiv 1 \pmod{m}$ thì ta lại có cách xử lý khá toán học kết hợp với thuật toán Pollard's rho để giảm ĐPT còn $\mathcal{O}\left(\sqrt[4]{m} \log m\right)$.
- Ngoài ra, bạn đọc tham khảo thêm một số thuật toán cải tiến:
  - Modulo hợp số:
    - Thuật toán Pohlig-Hellman (ĐPT là $\mathcal{O}(\sum \sqrt{p_i})$ với p_i là nằm trong phân tích hàm phi Euler : $\varphi(m) = p_1 \, p_2 \ldots p_r$).
  - Modulo nguyên tố:
    - Thuật toán Index Calculus - ĐPT $\mathcal{O}\left(e^{\sqrt{\log m \log \log m}}\right)$
    - Function field sieve - ĐPT $\mathcal{O}\left(e^{\sqrt[3]{\log m (\log \log m)^2}}\right)$
  - Tham khảo code C++ sử dụng thuật toán *Index Calculus* và *Pohlig-Hellman* tại đây .
- Logarit rời rạc đến nay vẫn là một bài toán khó, thường được sử dụng để xây dựng cấu trúc cho một hệ mật mã chẳng hạn như mật mã ElGamal , Trao đổi khoá Diffie-Hellman , Chữ ký số Elgamal , ...

Bài tập luyện tập

- Codeforces - Discrete Logarithm is a Joke
- Spoj - Power Modulo Inverted
- VNOJ - Dytechlab Algorithms Battle - Số dư
- Topcoder - SplittingFoxes3
- CodeChef - Inverse of a Function
- Hard Equation
- CodeChef - Chef and Modular Sequence

Tài liệu tham khảo

- CP-Algorithms - Discrete Log

- Wikipedia - Discrete logarithm