Discrete Logarithm Algorithms
Description and Implementation

Davide Bergamaschi

2019

Abstract

In this document we present the cryptographic problem of discrete logarithm and a number of algorithms to solve it, along with our implementation of said algorithms in the Python programming language. In particular, the following algorithms will be examined: exhaustive search, Baby-Step Giant-Step, Pollard's Rho, Pohlig-Hellman.

1 The Discrete Logarithm Problem

Let G be a cyclic group of order n, let α be a generator of G and let β belong to G. The discrete logarithm of β to the base α ($\log_\alpha \beta$) is the integer value x such that $0 \leq x \leq n-1$ and $\alpha^x = \beta$.

2 Solving Discrete Logarithms

Calculating discrete logarithms is in general a computationally hard problem. In the following section we present a selection algorithms that perform this task with different levels of sophistication.

Each algorithm is implemented by a Python function which takes α, β and n as its first arguments, and returns either $\log_\alpha \beta$ or $None$ if the logarithm cannot be found due to input data not respecting preconditions (e.g. if α is not a generator) or in case of failure of a randomized algorithm.

2.1 Exhaustive Search

This naive method consists in computing $\alpha^0, \alpha^1, \ldots, \alpha^{n-1}$ until β is eventually obtained.

Python Implementation 1: Exhaustive Search

```python
def exhaustive(alpha, beta, n):
    exp = 0
    power = alpha ** 0

    while power != beta:
        exp += 1

        if exp == n:
            return None

        power *= alpha

    return exp
```

The time complexity is trivially given by $O(n)$ multiplications.

2.2 Baby-Step Giant-Step

This method relies on the fact that the logarithm x can be written as $x = im + j$, where m can be conveniently chosen as $\lceil \sqrt{n} \rceil$. By doing so, one obtains $\beta = \alpha^x = \alpha^{im+j} = \alpha^{im}\alpha^j$, which is true if and only if $\beta(\alpha^{-m})^i = \alpha^j$.

The algorithm hence proceeds in the following way. For $0 \leq j < m$, entries (a^j, j) are computed and stored in a hash table (hashed on the first component).

Then, for $0 \leq i < \lceil n/m \rceil$, $\beta(\alpha^{-m})^i$ is computed. At each iteration the algorithm checks if the obtained value is present in the hash table. Upon finding a match with a value j, the logarithm x is obtained as $x = im + j$.

Python Implementation 2: Baby-Step Giant-Step

```python
def babygiant(alpha, beta, n, m=None):
    if m is None:
        m = round_sqrt(n)

    exp_table = {}

    power = alpha ** 0
    exp_table[power] = 0
    for j in range(1, m):
        power *= alpha
        if power not in exp_table:
            exp_table[power] = j

    factor = alpha ** (-m)
    if factor is None:
        return None

    candidate = beta
    for i in range((n + m - 1) // m):
        if candidate in exp_table:
            return i * m + exp_table[candidate]

        candidate *= factor

    return None
```

In the first part, if `m` has not been specified, it is calculated as $\lceil\sqrt{n}\rceil$. The lookup table is then constructed accordingly.

In each iteration of the last part, $\beta(\alpha^{-m})^i$ is computed by multiplying the `candidate` variable, initially set to β, by the precomputed value α^{-m}.

The algorithm performs $O(m)$ group multiplications while constructing the table, and $O(n/m)$ multiplications and lookups in the last part. If m is set to $\sqrt{n}$, the time complexity becomes $O(\sqrt{n})$. The space complexity is instead given by the m group elements stored by the algorithm.

2.3 Pollard's Rho Algorithm for Logarithms

This randomized algorithm explores a sequence of group elements $\gamma_i = \alpha^{a_i}\beta^{b_i}$ looking for a cycle using Floyd's algorithm.

Detecting a cycle means discovering two elements $\gamma_c = \alpha^a\beta^b$ and $\gamma_{2c} = \alpha^A\beta^B$ in the sequence, such that $\gamma_c = \gamma_{2c}$.

It follows that $\alpha^a\beta^b = \alpha^A\beta^B$, and so $\beta^{b-B} = \alpha^{a-A}$. By taking the logarithm to the base of α of both sides of the equation one obtains the following equation:

$$(b - B)\log_\alpha \beta \equiv (a - A) \pmod{n},$$

which can be solved to find $x = \log_\alpha \beta$, but only if $b \not\equiv B \pmod{n}$.

Let us discuss how the sequence of group elements is generated. To increase the likelihood of obtaining a cycle in a small number of samples, G is partitioned into three disjoint subsets S_0, S_1 and S_2 of approximately equal size and chosen in a sufficiently "random" manner.

The map for group elements $f : G \to G$ is then defined as:

$$f(\gamma) = \begin{cases} \beta\gamma & \text{if } \gamma \in S_0 \\ \gamma^2 & \text{if } \gamma \in S_1 \\ \alpha\gamma & \text{if } \gamma \in S_2 \end{cases},$$

while the map for a_i coefficients $g : G \times \mathbb{N} \to \mathbb{N}$ and the map for b_i coefficients $h : G \times \mathbb{N} \to \mathbb{N}$ are consequently defined as:

$$g(\gamma, a) = \begin{cases} a & \text{if } \gamma \in S_0 \\ (2a) \bmod n & \text{if } \gamma \in S_1 \\ (a+1) \bmod n & \text{if } \gamma \in S_2 \end{cases}, \qquad h(\gamma, b) = \begin{cases} (b+1) \bmod n & \text{if } \gamma \in S_0 \\ (2b) \bmod n & \text{if } \gamma \in S_1 \\ b & \text{if } \gamma \in S_2 \end{cases}.$$

The algorithm begins by iterating over the sequence induced by the above maps, starting from two random coefficients a_0 and b_0 (and with $x_0 = \alpha^{a_0}\beta^{b_0}$); in particular, at each step i it computes:

$$\begin{aligned} \gamma_i &= f(\gamma_{i-1}) & a_i &= g(\gamma_{i-1}, a_{i-1}) & b_i &= h(\gamma_{i-1}, a_{i-1}) \\ \gamma_{2i} &= f(\gamma_{2i-2}) & a_{2i} &= g(\gamma_{2i-2}, a_{2i-2}) & b_{2i} &= h(\gamma_{2i-2}, b_{2i-2}), \end{aligned}$$

and compares the obtained γ_i and γ_{2i} values.

When the algorithm finds $\gamma_i = \gamma_{2i}$, that is when a cycle is detected, if $(b_i - b_{2i}) \bmod n \neq 0$ the algorithm returns the logarithm:

$$x = (b_i - b_{2i})^{-1}(a_{2i} - a_i) \bmod n,$$

otherwise it terminates with failure.

Python Implementation 3: Pollard's Rho Algorithm for Logarithms

```
_A_MAP = [
    lambda a, n: a,
    lambda a, n: (a * 2) % n,
    lambda a, n: (a + 1) % n
]

_B_MAP = [
    lambda b, n: (b + 1) % n,
    lambda b, n: (b * 2) % n,
    lambda b, n: b
]

_F_MAP = [
    lambda gamma, alpha, beta: gamma * beta,
    lambda gamma, alpha, beta: gamma ** 2,
    lambda gamma, alpha, beta: gamma * alpha
]

def _map(a, b, gamma, alpha, beta, n, s_num):
    a = _A_MAP[s_num](a, n)
    b = _B_MAP[s_num](b, n)
    gamma = _F_MAP[s_num](gamma, alpha, beta)
    return (a, b, gamma)

def pollard(alpha, beta, n, s_map, a_start=0, b_start=0):
    a_slow = a_fast = a_start
    b_slow = b_fast = b_start
    gamma_slow = gamma_fast = (alpha ** a_start) * (beta ** b_start)

    while True:
        a_slow, b_slow, gamma_slow = _map(
            a_slow, b_slow, gamma_slow, alpha, beta, n, s_map(gamma_slow)
        )

        for _ in range(2):
            a_fast, b_fast, gamma_fast = _map(
                a_fast, b_fast, gamma_fast, alpha, beta, n, s_map(gamma_fast)
            )

        if gamma_slow == gamma_fast:
            r = (b_slow - b_fast) % n

            if r == 0:
                return None

            r_inv = mod_inverse(r, n)

            if r_inv is None:
                return None

            return (r_inv * (a_fast - a_slow)) % n
```

The `s_map` argument is a function which takes a group element as input and returns the index of the partition to which the element belongs.

For what concerns asymptotic complexity, if it can be assumed that the map f behaves like a random function, the expected running time will be of $O(\sqrt{n})$ group operations. On the other hand, the required storage is negligible.

2.4 Pohlig-Hellman Algorithm

This method leverages the prime factorization of the group order: $n = p_1^{e_1} \dots p_r^{e_r}$.

For $1 \leq i \leq r$, the algorithm calculates $x_i = x \bmod p_i^{e_i}$, obtaining a set of congruences that can ultimately be solved with the Chinese Remainder Theorem to find x.

In particular each x_i can be seen as a truncated p_i-ary representation of x:

$$x_i = x \bmod p_i^{e_i} = l_0 + l_1 p_i + l_2 p_i^2 + \dots + l_{e_i - 1} p_i^{e_i - 1},$$

which allows for a further reduction of the magnitude of the calculations.

Let us examine how the digits are calculated.

From the definition of modulus, there exists some integer h for which $x = x_i + h p_i^{e_i}$. Since $\beta = \alpha^x$, one can derive:

$$\begin{aligned}
\beta^{n/p_i} &= (\alpha^x)^{n/p_i} \\
&= (\alpha^{n/p_i})^x \\
&= (\alpha^{n/p_i})^{x_i + h p_i^{e_i}} \\
&= (\alpha^{n/p_i})^{l_0 + l_1 p_i + l_2 p_i^2 + \dots + l_{e_i - 1} p_i^{e_i - 1} + h p_i^{e_i}} \\
&= (\alpha^{n/p_i})^{l_0 + K p_i} \quad \text{(for some integer } K) \\
&= (\alpha^{n/p_i})^{l_0} (\alpha^{n/p_i})^{K p_i} \\
&= (\alpha^{n/p_i})^{l_0} \quad \text{(since } \alpha^{n/p_i} \text{ is of order } p_i),
\end{aligned}$$

from which l_0 can be found using any other discrete logarithm technique.

Then, starting from $\beta\alpha^{-l_0} = \alpha^{l_1 p_i + l_2 p_i^2 + \dots + l_{e_i - 1} p_i^{e_i - 1} + h p_i^{e_i}}$ and by raising both sides to the power of ${}^{n}/{}_{p_i^2}$, one can obtain $(\alpha^{n/p_i})^{l_1} = (\beta\alpha^{-l_0})^{n/p_i^2}$ and ultimately l_1 in a similar way.

The remaining digits can be computed analogously.

The calculations performed for each $p_i^{e_i}$ in the first part of the algorithm are in conclusion the following:

$$\begin{aligned}
l_0 &= \log_{\alpha^{n/p_i}} (\beta)^{n/p_i} \\
l_1 &= \log_{\alpha^{n/p_i}} (\beta\alpha^{-l_0})^{n/p_i^2} \\
l_2 &= \log_{\alpha^{n/p_i}} (\beta\alpha^{-l_0 - l_1 p_i})^{n/p_i^3} \\
&\vdots \\
l_{e_i - 1} &= \log_{\alpha^{n/p_i}} (\beta\alpha^{-l_0 - l_1 p_i - \dots - l_{e_i - 2} p_i^{e_i - 2}})^{n/p_i^{e_i}}.
\end{aligned}$$

Determining all the x_i values yields a set of congruences of the following type:

$$\begin{aligned} x &\equiv x_1 \pmod{p_1^{e_1}} \\ &\vdots \\ x &\equiv x_r \pmod{p_r^{e_r}}. \end{aligned}$$

Since the moduli are pairwise coprime by construction, the Chinese Remainder Theorem guarantees the existence of a solution x, which can be found with any suitable computational method (e.g. Gauss's algorithm).

Python Implementation 4: Pohlig-Hellman Algorithm

```python
def pohlighellman(alpha, beta, n, n_factors):
    remainders = []
    moduli = []

    for p, e in n_factors.items():
        base = alpha ** (n // p)

        arg_base = beta
        cur_pow = 0
        next_pow = 1
        l = 0
        rem = 0

        for j in range(e):
            prev_l = l

            prev_pow = cur_pow
            cur_pow = next_pow
            next_pow *= p

            arg_base *= alpha ** (- prev_l * prev_pow)
            arg_exp = n // next_pow
            arg = arg_base ** arg_exp

            l = exhaustive(base, arg, n)

            if l is None:
                return None

            rem += l * cur_pow

        remainders.append(rem)
        moduli.append(p ** e)

    return crt(moduli, remainders)
```

The prime factorization of n is passed as fourth argument in the form of a Python dictionary, with keys being the prime factors and values are respectively their exponents.

Notice that during the computation of an x_i value, at each iteration j in the internal loop the implementation keeps track of the necessary powers of p_i (p_i^{j-1}, p_i^j and p_i^{j+1}), so that only one multiplication is enough to calculate them at each step.

The running time is given by $O(\sum_{i=1}^{r} e_i(\lg n + \sqrt{p_i}))$ group multiplications.