# GROUP CRYPTOGRAPHY AND DISCRETE LOGARITHMS

Enes KALECI

MATH 411 Capstone Project 2

**Advisor:** Doga Can Sertbas

Istinye University
Faculty of Engineering and Natural Sciences
Mathematics Department

May 2025

# GROUP CRYPTOGRAPHY AND DISCRETE LOGARITHMS

Enes KALECI

28/05/2025

**Abstract**

This study investigates the basic uses of group theory in encryption and the importance of the discrete logarithm problem in cryptographic security. The first phase covers the ElGamal encryption algorithm, followed by the presentation of key strategies for solving the discrete logarithm problem: the baby-step giant-step method, the Pohlig-Hellman algorithm, and the index calculus method. Other mathematical tools, such as the Chinese Remainder Theorem, that aid in the acceleration of systems based on discrete logarithms, as well as cryptographic attacks like birthday attacks, are examined.

In phase two, the theoretical concepts were implemented as practical models within the SageMath software. The algorithms of interest were created and executed using SageMath in order to demonstrate how the operational heuristics could be inspected to showcase the power of computer algebra systems in cryptographic tasks.

**Acknowledgement**
I would like to thank my Instructor, Doga Can Sertbas, for the guidance and support he provided throughout this study.

Contents

# 1 The ElGamal Encryption Scheme

## ElGamal Encryption Protocol:

- **Set-up:** Choose a finite cyclic group $G = \langle g \rangle$ with $d$ elements. Let $g$ be a generator. These are made public.
- **Key Generation:**
  - Private key: $b \in \mathbb{Z}_d$
  - Public key: $B = g^b \in G$
- **Encryption:**
  - Input: plaintext $x \in G$
  - Choose random $a \in \mathbb{Z}_d$
  - Compute $A = g^a$, $k = B^a$, $y = x \cdot k$
  - Output: ciphertext $(y, A)$
- **Decryption:**
  - Input: ciphertext $(y, A)$
  - Compute $k = A^b$, then $x = y \cdot k^{-1}$

Below the code of the protocol is written using sagemath and is mentioned below.

**Python / SageMath Implementation:**

```python
p = random_prime(2^11, 2^12)
G = GF(p)
g = G.primitive_element()
print(f"Prime number (p): {p}")
print(f"Generator (g): {g}")

# Key Generation
b = randint(0, p-1)
B = g^b
print(f"Private Key (b): {b}")
print(f"Public Key (B): {B}")

# Message input
x_input = input(f"Enter a number in GF({p}) (x): ")
x = G(int(x_input))

# Encryption Function
def encrypt(x, B, g, p):
    a = randint(0, p-1)
    A = g^a
    k = B^a
    y = x * k
    return (y, A)

```

```
ciphertext = encrypt(x, B, g, p)
print(f"A: {ciphertext[1]}")
print(f"y: {ciphertext[0]}")
print(f"Ciphertext (y, A): {ciphertext}")

# Decryption Function
def decrypt(y, A, b, p):
    k = A^b
    k_inv = k^(-1)
    print(f"Shared secret (k): {k}")
    print(f"Inverse of k: {k_inv}")
    return y * k_inv

decrypted_x = decrypt(ciphertext[0], ciphertext[1], b, p)
print(f"Decrypted Message: {decrypted_x}")
```

Listing 1: ElGamal Encryption Example in SageMath

## 2 Baby-step Giant-step Algorithm

**Algorithm 1.** Baby-step Giant-step for discrete logarithms

**Input:**

- A cyclic group $G = \langle g \rangle$ with order $d$
- Element $x \in G$

**Output:** $\log_g x \in \mathbb{Z}_d$

1. Set $m = \lceil \sqrt{d} \rceil$
2. Compute and store: $xg^j \mod n$ for $j = 0$ to $m - 1$
3. Compute $g^m$, then $g^{im} \mod n$ for $i = 1$ to $m$
4. If a match is found, return $a = im - j$

The code for the Baby-Step Giant-Step Algorithm was written using sagemath and gives the code.

**Python / SageMath Implementation:**

```
from math import ceil, sqrt
from math import gcd

def BSGS(g, x, n, order):
    if gcd(g, n) != 1 or gcd(x, n) != 1:
        print("ERROR: g and x must be coprime with n.")
        return None

    m = ceil(sqrt(order))
    baby_steps = {}

    for j in range(m):
        val = (x * pow(g, j, n)) % n
        baby_steps[val] = j
        print(f"Baby j={j}: x * g^{j}  {val} mod {n}")

    g_m = pow(g, m, n)
    for i in range(1, m + 1):
        y = pow(g_m, i, n)
        print(f"Giant i={i}: g^{i * m}  {y} mod {n}")
        if y in baby_steps:
            j = baby_steps[y]
            result = (i * m - j) % order
            print(f"Result: k = ({i} * {m} - {j}) % {order} = {result}")
            return result

    print("No solution found.")
    return None


n = 25
g = 2
x = 17
```

```
order = 20
k = BSGS(g, x, n, order)
```

Listing 2: Baby-step Giant-step Implementation

## 3 Birthday Algorithm for Discrete Logarithm

**Algorithm 2.** Birthday Algorithm for Discrete Logarithm

**Input:** A cyclic group $G = \langle g \rangle$ with $d$ elements, and a group element $x \in G$.

**Output:** $\log_g x \in \mathbb{Z}_d$.

1. Initialize sets $X \leftarrow \emptyset$, $Y \leftarrow \emptyset$.
2. Repeat until a collision occurs:
   - Randomly choose $b \in \{0, 1\}$, and $i \in \{0, \ldots, d-1\}$
   - If $b = 0$: compute $xg^i$, store in $X$
   - If $b = 1$: compute $g^i$, store in $Y$
3. If a collision $xg^i = g^j$ is found: return $j - i \mod d$

The code for the Birthday Algorithm was written using sagemath and gives the code.

**Python / SageMath Implementation:**

```python

def birthday_algorithm(g, x, p):
    d = totient(p)  # Euler's totient function
    X, Y = {}, {}   # Hash tables for collision detection

    while True:
        b = randint(0, 1)
        i = randint(0, d - 1)

        if b == 0:
            value = (x * pow(g, i, p)) % p
            if value in Y:
                return (Y[value] - i) % d
            X[value] = i
        else:
            value = pow(g, i, p)
            if value in X:
                return (i - X[value]) % d
            Y[value] = i

# User input
p = int(input("Modular number p (prime): "))
G = GF(p)
g = G.primitive_element()
print(f"Primitive element over GF({p}): {g}")
x = int(input("Encrypted message x: "))

dlog = birthday_algorithm(g, x, p)
print(f"\nDiscrete logarithm (in Z{p}*): {dlog}")
check = pow(int(g), dlog, p)
print(f"Verification: g^a mod p = {check}, x = {x} (should be equal)")
```

Listing 3: Birthday Algorithm Implementation

## 4 Chinese Remaindering for Discrete Logarithms

**Algorithm Description:** The Chinese remaindering method for discrete logarithms is used when the group order $d$ can be factored into small prime components. The discrete logarithm problem is solved separately modulo each prime factor, and these smaller solutions are then combined using the Chinese Remainder Theorem to obtain the overall solution.

**Algorithm 3.** Chinese remaindering for discrete logarithms.
**Input:** A cyclic group $G = \langle g \rangle$ of order $d = \#G$, and $x \in G$.
**Output:** $a = \mathrm{dlog}_g x$.

1. Compute the prime power factorization of $d$.
2. For each $i \leq r$, do steps 3 and 4:
   (a) Compute $g_i = g^{d/q_i}$ and $x_i = x^{d/q_i}$, using the Repeated Squaring Algorithm
   (b) Compute the discrete logarithm $a_i = \mathrm{dlog}_{g_i} x_i \in \mathbb{Z}_{q_i}$ in $S_i = \langle g_i \rangle$.
3. Combine these "small" discrete logarithms via the Chinese Remainder Algorithm to find the unique $a \in \mathbb{Z}_d$ such that $a = a_i$ in $\mathbb{Z}_{q_i}$ for all $i \leq r$.

The code for the Chinese Remaindering Algorithm was written using sagemath and gives the code.

**Python / SageMath Implementation:**

```

def dlog(g, x, n, order):
    if gcd(g, n) != 1 or gcd(x, n) != 1:
        print("ERROR: g and x must be coprime with n!")
        return None

    m = ceil(sqrt(order))
    baby_steps = {}

    for j in range(m):
        val = (x * pow(g, j, n)) % n
        baby_steps[val] = j

    g_m = pow(g, m, n)
    for i in range(1, m + 1):
        y = pow(g_m, i, n)
        if y in baby_steps:
            j = baby_steps[y]
            result = (i * m - j) % order
            print(f"Result: k = ({i} * {m} - {j}) % {order} = {result}")
            return result

    print("No solution found.")
```

```
    return None

def chinese_remainder_dlog(p, x):
    print(f"p = {p}, x = {x}")
    d = euler_phi(p)
    print(f"Group order d = ({p}) = {d}")
    g = find_primitive_root(p)
    print(f"Primitive root: g = {g}")
    factors = factor(d)
    print(f"Factorization of d: {factors}")

    moduli, congruences = [], []

    for prime_factor, exponent in factors:
        q_i = prime_factor ** exponent
        print(f"\nProcessing for q_i = {q_i}:")
        g_i = power_mod(g, d // q_i, p)
        x_i = power_mod(x, d // q_i, p)
        print(f"g_i = {g_i}, x_i = {x_i}")
        a_i = dlog(g_i, x_i, p, q_i)
        if a_i is None:
            print(f"No solution for q_i = {q_i}")
            return None
        print(f"Found a_i = {a_i} mod {q_i}")
        moduli.append(q_i)
        congruences.append(a_i)

    a = crt(congruences, moduli)
    print(f"\nResult using CRT: a = {a} mod {d}")
    check = power_mod(g, a, p)
    print(f"Verification: g^a mod p = {check} (should be x = {x})")
    return a

def find_primitive_root(p):
    d = euler_phi(p)
    factors = [pf[0] for pf in factor(d)]
    for g in range(2, p):
        if gcd(g, p) != 1:
            continue
        ok = all(power_mod(g, d // q, p) != 1 for q in factors)
        if ok:
            return g
    raise ValueError("No primitive root found.")

# Example usage
p = 179424673
x = 225
a = chinese_remainder_dlog(p, x)
print(f"Final result: a = {a}")
```

Listing 4: Chinese Remaindering for Discrete Logarithms

## 5 The Pohlig-Hellman Algorithm

**Algorithm Description:**This algorithm works for groups of order $p^e$.

**Algorithm 4.** Pohlig-Hellman Algorithm
**Input:** A cyclic group $G = \langle g \rangle$ of order $p^e$, where $p$ is a prime, $e \geq 2$ is an integer, and $x \in G$.
**Output:** $a = \mathrm{dlog}_G(x)$, the discrete logarithm of $x$ in $G$.

1. Compute $h = g^{p^{e-1}}$ and set $y_{-1} = 1 \in G$.
2. For $i = 0$ to $e - 1$, do:
   (a) $x_i \leftarrow (x \cdot y_{i-1}^{-1})^{p^{e-i-1}}$
   (b) Compute $a_i \leftarrow \mathrm{dlog}_h(x_i)$.
   (c) Update $y_i \leftarrow y_{i-1} \cdot g^{-a_i p^i}$.
3. Return $a = a_{e-1}p^{e-1} + \cdots + a_0$.

The code for the Pohlig hellman Algorithm was written using sagemath and gives the code.

**Python / SageMath Implementation:**

```
from math import ceil, sqrt

def dlog(g, x, n, order):
    if gcd(g, n) != 1 or gcd(x, n) != 1:
        print("ERROR: g and x must be coprime with n!")
        return None

    m = ceil(sqrt(order))
    baby_steps = {}

    for j in range(m):
        val = (x * power_mod(g, j, n)) % n
        baby_steps[val] = j

    g_m = power_mod(g, m, n)
    for i in range(1, m + 1):
        y = power_mod(g_m, i, n)
        if y in baby_steps:
            j = baby_steps[y]
            result = (i * m - j) % order
            return result

    print("No solution found.")
    return None

def pohlig_hellman(F, g, x, p_1, d_1):
    prime = Integer(p_1)
    e = int(d_1)
    pe = prime ^ e

    print(f"\np^e = {pe} and its prime factorization: {factor(pe)}")
```

```

    if (x ^ pe) != 1:
        raise ValueError(f"The given x is not an element of the subgroup of order p^e (pe = {pe})! Please choose a valid x.")

    h = F(g) ^ (prime ^ (e - 1))
    y_prev = F(1)
    a_values = []

    for i in range(e):
        power = prime ^ (e - i - 1)
        x_i = (x * y_prev) ^ power
        print(f"i={i}: x_i = (x * y_prev)^{power} = {x_i}")

        order_h = h.multiplicative_order()
        a_i = dlog(Integer(h), Integer(x_i), F.order(), order_h)
        print(f"a_{i} = log_h({int(x_i)}) = {a_i}")
        a_values.append(a_i)

        y_prev = y_prev * (F(g) ^ (-a_i * (prime ^ i)))
        print(f"y_{i} updated = {y_prev}")

    result = sum(a_values[i] * (prime ^ i) for i in range(e))
    print(f"\nResult (mod {pe}): {result}")
    return result

def create_subgroup(F, g, p_1, d_1, p):
    n = p_1 ^ d_1
    phi_p = euler_phi(p)
    g_0 = power_mod(g, phi_p // n, p)
    print(f"Subgroup Generator g_0 = {g_0}")

    subgroup = [power_mod(g_0, i, p) for i in range(n)]
    print(f"Subgroup: {subgroup}")
    return g_0, subgroup

# Main program
p = Integer(input("Enter a prime number P: "))
if not is_prime(p):
    print("The entered number is not prime.")
    quit()

F = GF(p)
phi_p = euler_phi(p)
factors = factor(phi_p)

if all(e < 2 for _, e in factors):
    print("None of the exponents are 2 or more, algorithm will not work.")
    quit()

g = F.multiplicative_generator()
print(f"Generator g = {g}")

print("Choose a factor:")
options = {}
for f, e in factors:
    if e >= 2:
        val = f ^ e
        options[str(val)] = (f, e)
        print(f"{f}^{e} = {val}")

choice = input("Your choice: ")
if choice not in options:
```

```
    print("Invalid selection!")
    quit()

p_1, d_1 = options[choice]

g_0, subgroup = create_subgroup(F, g, p_1, d_1, p)

x = Integer(input("Enter the target number x: "))

try:
    result = pohlig_hellman(F, g_0, F(x), p_1, d_1)
    print(f"\nSolution: x = {result} (mod {p_1 ^ d_1})")
except Exception as e:
    print(f"Error: {e}")
```

Listing 5: Pohlig-Hellman Algorithm Implementation

## 6 Discrete logarithm in a cyclic group

**Algorithm 5.** Discrete Logarithm in a Cyclic Group (Pohlig-Hellman General Case)
**Input:** A finite cyclic group $G = \langle g \rangle$ of order $d$, and an element $x \in G$.
**Output:** $a = \log_g x$, the discrete logarithm of $x$ in $G$.

1. Factor the group order $d$ as $d = p_1^{e_1} \cdots p_r^{e_r}$ with distinct primes $p_i$ and positive integers $e_i$.
2. For $i = 1$ to $r$, do:
   (a) Let $q_i = p_i^{e_i}$, and compute $g_i = g^{d/q_i}$, $x_i = x^{d/q_i}$. These lie in the subgroup $\langle g_i \rangle$ of order $q_i$.
   (b) Compute the discrete logarithm $a_i = \log_{g_i}(x_i)$:
      - Use the Pohlig-Hellman algorithm recursively if $e_i > 1$,
      - Use Baby-step Giant-step Algorithm if $e_i = 1$.
3. Combine the results $a_i \mod q_i$ using the Chinese Remainder Theorem to find $a \mod d$.

### Python / SageMath Implementation:

```
# For prime powers p^e e hem 1 hem 1 den büyük için
def dlog(g, x, n, order):
    if gcd(g, n) != 1 or gcd(x, n) != 1:
        print("ERROR: g and x must be coprime with n!")
        return None

    m = ceil(sqrt(order))
    baby_steps = {}

    for j in range(m):
```

```
        val = (x * power_mod(g, j, n)) % n
        baby_steps[val] = j

    g_m = power_mod(g, m, n)
    for i in range(1, m + 1):
        y = power_mod(g_m, i, n)
        if y in baby_steps:
            j = baby_steps[y]
            result = (i * m - j) % order
            print(f"  -> dlog solution: i={i}, j={j} → x = {result}")
            return result

    print("No solution found.")
    return None

def pohlig_hellman(F, g, x, p, e):
    print(f"\n[Starting Pohlig-Hellman] p = {p}, e = {e}")
    pe = p ^ e
    h = F(g) ^ (p ^ (e - 1))
    y_prev = F(1)
    a_values = []

    for i in range(e):
        power = p ^ (e - i - 1)
        x_i = (x * y_prev) ^ power
        order_h = h.multiplicative_order()

        print(f"\n  [Step {i}]")
        print(f"    x_i = (x * y_prev)^{power} = {x_i}")
        print(f"    h = g^{p}^{e - 1} = {h}")
        print(f"    Order of h: {order_h}")

        a_i = dlog(int(h), int(x_i), F.order(), order_h)
        print(f"    a_{i} = log_h({int(x_i)}) = {a_i}")
        a_values.append(a_i)

        y_prev *= F(g) ^ (-a_i * (p ^ i))
        print(f"    Updated y_{i} = {y_prev}")

    x_mod_pe = sum(a * (p ^ i) for i, a in enumerate(a_values))
    print(f"\n  --> x  {x_mod_pe} mod {pe}")
    return x_mod_pe

def generate_subgroup(g, x, p, e, mod_p):
    n = p ^ e
    phi = euler_phi(mod_p)
    g_0 = power_mod(g, phi // n, mod_p)
    x_0 = power_mod(x, phi // n, mod_p)
    print(f"\n[Generating Subgroup] {p}^{e}")
    print(f"  g_0 = g^({phi} / {n}) mod {mod_p} = {g_0}")
    print(f"  x_0 = x^({phi} / {n}) mod {mod_p} = {x_0}")
    return g_0, x_0

# Main Program
p = Integer(input("Enter a prime number P: "))
if not is_prime(p):
    print("ERROR: The number entered is not a prime.")
    quit()

F = GF(p)
phi_p = euler_phi(p)
factors = factor(phi_p)
print(f"\nphi({p}) = {phi_p} = {factors}")
```

```

g = F.multiplicative_generator()
print(f"\nGenerator g = {g}")

x = Integer(input("\nEnter the target value x: "))

mod_list = []
res_list = []

for prime, exponent in factors:
    mod = prime ^ exponent
    g_0, x_0 = generate_subgroup(g, x, prime, exponent, p)

    if exponent >= 2:
        print(f"Applying Pohlig-Hellman for {prime}^{exponent}...")
        res = pohlig_hellman(F, F(g_0), F(x_0), prime, exponent)
    else:
        print(f"Applying classic dlog for {prime}^{exponent}...")
        res = dlog(g_0, x_0, p, prime)

    if res is None:
        print("No solution found.")
        quit()

    mod_list.append(mod)
    res_list.append(res)

# Combine results using Chinese Remainder Theorem
x_final = crt(res_list, mod_list)
print(f"\nFinal Result: x  {x_final} mod {phi_p}")
```

Listing 6: Pohlig-Hellman General Case

## 7 Index Calculus Algorithm

**Algorithm 6.** Index Calculus Algorithm for Discrete Logarithm
**Input:** A prime $p$, a generator $g \in \mathbb{Z}_p^*$, and an element $x \in \mathbb{Z}_p^*$.
**Output:** $a = \log_g x \mod (p-1)$

1. **Factor Base Selection:**
   - Choose a smoothness bound $B$, and let $\mathcal{F} = \{p_1, \dots, p_h\}$ be the set of all prime numbers $\leq B$.
2. **Relation Collection:**
   - Repeat until $h + 20$ B-smooth relations are collected:
     (a) Randomly choose $e \in \mathbb{Z}_{p-1}$.
     (b) Compute $y = g^e \mod p$.
     (c) If $y$ is $B$-smooth over $\mathcal{F}$, i.e.,
     $$g^e \equiv p_1^{\alpha_1} \cdots p_h^{\alpha_h} \mod p,$$
     then we get the linear equation:
     $$e \equiv \alpha_1 \log_g p_1 + \cdots + \alpha_h \log_g p_h \mod (p-1).$$
3. **Solve Linear System:**
   - Use the collected relations to form a system of linear equations.
   - Solve this system modulo $p-1$ to find $\log_g p_1, \dots, \log_g p_h$.
4. **Compute $\log_g x$:**
   - Repeat:
     (a) Randomly choose $e \in \mathbb{Z}_{p-1}$.
     (b) Compute $y = x \cdot g^e \mod p$.
     (c) If $y$ is $B$-smooth:
     $$x \cdot g^e \equiv p_1^{\beta_1} \cdots p_h^{\beta_h} \mod p,$$
     then:
     $$\log_g x \equiv -e + \beta_1 \log_g p_1 + \cdots + \beta_h \log_g p_h \mod (p-1).$$
   - Return $\log_g x$.

```

p = 31
g = 3
h = 13
F = GF(p)
B = ceil(log(p)^2)
factor_base = list(primes(B + 1))

print(f"Generator used for GF({p}): g = {g}")
print(f"Factor base upper bound B = {B}")
print(f"Factor base ({len(factor_base)} primes): {factor_base}")

# === B-smooth check ===
def is_B_smooth(n, factor_base):
    try:
        factors = factor(n)
        for prime, _ in factors:
            if int(prime) not in factor_base:
                return False, []
        flat = []
        for prime, exp in factors:
            flat.extend([int(prime)] * exp)
        return True, flat
    except:
        return False, []

def factorlist_to_dict(factor_list):
    return dict(Counter(factor_list))

def collect_relations(g, p, factor_base, num_relations):
    relations = []
    used_exponents = set()
    print(f"\nCollecting {num_relations} B-smooth relations...")

    while len(relations) < num_relations:
        e = randint(2, p - 2)
        if e in used_exponents:
            continue
        used_exponents.add(e)
        val = power_mod(g, e, p)
        is_smooth, flat = is_B_smooth(val, factor_base)
        if is_smooth:
            exp_dict = factorlist_to_dict(flat)
            relations.append((exp_dict, e))

            primes_exp_str = ' * '.join([f"{prime}^{exp_dict[prime]}" for prime in sorted(exp_dict.keys())])
            log_terms = ' + '.join([f"{exp_dict[prime]}·log_g({prime})" for prime in sorted(exp_dict.keys())])
            print(f"\nRelation {len(relations)}:")
            print(f"g^{e}  {val}  {primes_exp_str} mod {p}")
            print(f"{e}  {log_terms} mod {p - 1}")

    return relations

def build_matrix_system(relations, factor_base, p):
    rows = []
    b_vector = []

    print("\nConstructing matrix A and vector b...")
    print("Matrix A (coefficient matrix):")

    for i, (exp_dict, e) in enumerate(relations):
```

```
        row = [exp_dict.get(base, 0) for base in factor_base]
        rows.append(row)
        b_vector.append(e % (p - 1))
        print(f"Row {i+1}: {row}  |  e = {e % (p - 1)}")

    A = Matrix(Integers(p - 1), rows)
    b = vector(Integers(p - 1), b_vector)

    print("\nVector b:")
    print(list(b))

    return A, b

def solve_dlog_matrix(A, b):
    try:
        x = A.solve_right(b)
        return x
    except:
        print("Failed to solve the linear system.")
        return None

def compute_log_h(g, h, p, factor_base, dlogs):
    print("\nComputing log_g(h)...")
    for r in range(1, 10000):
        val = int(mod(h * power_mod(g, r, p), p))
        is_smooth, factors = is_B_smooth(val, factor_base)
        if is_smooth:
            beta = factorlist_to_dict(factors)
            log_h = sum(dlogs.get(base, 0) * exp for base, exp in beta.items())
            log_h = (log_h - r) % (p - 1)

            beta_str = " + ".join([f"{beta[b]}·log_g({b})" for b in sorted(beta.keys())])
            print(f"\nB-smooth found: h * g^{r}   {val} mod {p}")
            print(f"log_g(h)   {beta_str} - {r} mod {p - 1}")
            print(f"Result: log_g({h})   {log_h} mod {p - 1}")
            return log_h

    print("Could not compute log(h). Try a larger B or more attempts.")
    return None

def index_calculus():
    num_relations = len(factor_base)
    relations = collect_relations(g, p, factor_base, num_relations)
    A, b = build_matrix_system(relations, factor_base, p)
    solution = solve_dlog_matrix(A, b)

    if solution is None:
        print("Operation failed.")
        return

    dlogs = {base: solution[i] for i, base in enumerate(factor_base)}
    print("\nlog_g(pi) values:")
    for base in factor_base:
        print(f"log_g({base}) = {dlogs[base]}")

    log_h = compute_log_h(g, h, p, factor_base, dlogs)

    if log_h is not None:
        print(f"\nRESULT: log_g({h})   {log_h} mod {p - 1}")
        if power_mod(g, int(log_h), p) == h:
            print(f"Verification successful: {g}^{int(log_h)}   {h} mod {p}")
        else:
```

```
            print(f"Verification failed: {g}^{int(log_h)} mod {p} = {power_mod(g, int(log_h), p)}  {h}")

# === Run the algorithm ===
index_calculus()
```

Listing 7: Index calculus

## References

[1] Joachim von zur Gathen, *CryptoSchool*, Springer, 2015.