Implementation Details for Baby-Step Giant-Step Algorithm

Matthew Nunez

December 8, 2024

Implementation Details

This program solves the discrete logarithm problem for elliptic curves using the baby-step giant-step algorithm, with a clear structure to ensure clarity and reusability. The goal is to find an integer k such that $Q = [k]P$, where P is a generator point and Q is another point on the elliptic curve $E(\mathbb{F}_p)$, defined by the equation

$$y^2 = x^3 + ax + b \mod p.$$

The `verify_inputs` function checks that P and Q lie on a well-defined curve by validating the discriminant $4a^3+27b^2 \neq 0 \mod p$ and ensuring the points satisfy the curve equation. This step ensures the algorithm starts with valid parameters.

The `calculate_order` function determines the order n of P, essential for understanding the cyclic structure of the group and the range for k. To handle modular arithmetic efficiently, the `inverse_mod` function implements the extended Euclidean algorithm to compute modular inverses, critical for elliptic curve point addition.

The `elliptic_addition` function handles the addition of two points on the curve, covering edge cases such as the point at infinity or vertical lines. This function is central to both precomputing baby steps and iterating giant steps. The `baby_step_giant_step_setup` function precomputes baby steps up to $m = \lceil \sqrt{n} \rceil$, storing them in a dictionary for fast and efficient lookups. It also calculates the giant step offset $-mP$ to enable efficient iteration.

Finally, the `baby_step_giant_step_solve` function integrates all these components. It verifies inputs, computes the group order, and implements the algorithm by matching giant steps against precomputed baby steps. By structuring the program into these functions, I ensured error handling and a clear implementation of the algorithm. This approach guarantees efficiency while maintaining checks to validate inputs and computations.

Code and Results

The following section contains the code and results, embedded from an external PDF file.

corrected final - matthew n

December 11, 2024

[3]:
```
def is_prime(n):
    """check if a number is prime."""
    if n < 2:
        return False
    for i in range(2, int(sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True

def inverse_mod(a, m):
    """calculating the modular multiplicative inverse of a modulo m, this will␣
 ↪allow me to do the point addition much more easily"""
    def extended_gcd(a, b):
        if a == 0:
            return b, 0, 1
        gcd, x1, y1 = extended_gcd(b % a, a)
        x = y1 - (b // a) * x1
        y = x1
        return gcd, x, y

    gcd, x, _ = extended_gcd(a, m)
    if gcd != 1:
        raise ValueError("Modular inverse does not exist")
    return x % m

def verify_inputs(p, a, b, P, Q):
    """verifying input parameters for the elliptic curve and points"""
    if not is_prime(p):
        print("Error: p is not prime.")
        return False

    discriminant = (4 * a**3 + 27 * b**2) % p
    if discriminant == 0:
        print("Error: the discriminant is zero. the curve is singular.")
        return False

    u, v = P
```

```
    if (v**2 % p) != ((u**3 + a * u + b) % p):
        print(f"Error: P = {P} is not on the curve.")
        return False

    x, y = Q
    if (y**2 % p) != ((x**3 + a * x + b) % p):
        print(f"Error: Q = {Q} is not on the curve.")
        return False

    print("all inputs are valid.")
    return True

def elliptic_addition(P1, P2, a, b, p):
    """performing elliptic curve point addition for P1 + P2."""
    if P1 == (None, None):
        return P2
    if P2 == (None, None):
        return P1

    x1, y1 = P1
    x2, y2 = P2

    if x1 == x2:
        if (y1 + y2) % p == 0:
            return (None, None)
        try:
            slope = (3 * x1**2 + a) * inverse_mod(2 * y1, p) % p
        except ValueError:
            return (None, None)
    else:
        try:
            slope = ((y2 - y1) * inverse_mod((x2 - x1) % p, p)) % p
        except ValueError:
            return (None, None)

    x3 = (slope**2 - x1 - x2) % p
    y3 = (slope * (x1 - x3) - y1) % p

    return (x3, y3)

def calculate_order(p, a, b, P):
    """calculate the order of point P on the curve."""
    current = P
    order = 1

    while current != (None, None):
        current = elliptic_addition(current, P, a, b, p)
```

```
        order += 1
        if order > p + 1 + 2 * sqrt(p):  # hasse's theorem check
            return None

    return order

def baby_step_giant_step_setup(p, a, b, P, Q):
    """ baby-step giant-step algorithm setup function."""
    # calculate or estimate the order
    n = p + 1 + 2 * sqrt(p)  # hasses's theorem upper bound
    m = ceil(sqrt(n))

    # baby steps
    baby_steps = {}
    current = P  # start with P (not point at infinity)
    baby_steps[current] = 1  # store first point

    temp = P
    for j in range(2, m+1):
        temp = elliptic_addition(temp, P, a, b, p)
        if temp == (None, None):
            continue
        baby_steps[temp] = j

    # computing -m*P for giant steps
    mP = P
    for _ in range(m-1):
        mP = elliptic_addition(mP, P, a, b, p)
    neg_mP = (mP[0], (-mP[1]) % p) if mP != (None, None) else (None, None)

    return baby_steps, neg_mP, m

def baby_step_giant_step_solve(p, a, b, P, Q):
    """main function for solving the discrete logarithm."""
    if not verify_inputs(p, a, b, P, Q):
        return None

    print("step 1 completed: input verification successful.")

    order = calculate_order(p, a, b, P)
    if order is None:
        print("Error: could not calculate the order of P.")
        return None
    print(f"step 2 completed: the order of P is {order}.")

    baby_steps, neg_mP, m = baby_step_giant_step_setup(p, a, b, P, Q)
    print(f"step 3 completed: baby-step setup with m = {m}.")
```

```
    current = Q

    # computing giant steps
    for i in range(m):
        if current in baby_steps:
            j = baby_steps[current]
            k = i * m + j
            print(f"match found: i = {i}, j = {j}, k = {k}")
            print(f"step 4 completed: the discrete logarithm k is {k}.")
            return k
        current = elliptic_addition(current, neg_mP, a, b, p)

    print("step 4 completed: no solution found.")
    return None


# all test cases
test_cases = [
    {"p": 6917, "a": 0, "b": 6521, "P": (210, 1338), "Q": (3329, 6379)},
    {"p": 6047, "a": 0, "b": 5986, "P": (804, 2425), "Q": (4791, 105)},
    {"p": 5953, "a": 0, "b": 4687, "P": (841, 5852), "Q": (5577, 235)}
    ]

# solving each test case
for idx, case in enumerate(test_cases, start=1):
    print(f"\nrunning Test Case {idx}...")
    k = baby_step_giant_step_solve(case["p"], case["a"], case["b"], case["P"],␣
 ↪case["Q"])
    if k is not None:
        print(f"test case {idx}: final solution k = {k}\n")
    else:
        print(f"test case {idx}: no solution found.\n")
```

```
running Test Case 1…
all inputs are valid.
step 1 completed: input verification successful.

step 2 completed: the order of P is 6918.
step 3 completed: baby-step setup with m = 85.
match found: i = 63, j = 16, k = 5371
step 4 completed: the discrete logarithm k is 5371.
test case 1: final solution k = 5371


running Test Case 2…
all inputs are valid.
```

```
step 1 completed: input verification successful.

step 2 completed: the order of P is 6048.
step 3 completed: baby-step setup with m = 79.
match found: i = 63, j = 21, k = 4998
step 4 completed: the discrete logarithm k is 4998.
test case 2: final solution k = 4998


running Test Case 3…
all inputs are valid.
step 1 completed: input verification successful.

step 2 completed: the order of P is 5961.
step 3 completed: baby-step setup with m = 79.
match found: i = 59, j = 28, k = 4689
step 4 completed: the discrete logarithm k is 4689.
test case 3: final solution k = 4689
```

Submission Information

- **Name:** Matthew Nunez
- **Net ID:** mn3040
- **Course:** Number Theory and Cryptography (Fall 2024)
- **Instructor:** Arman Mimar
- **Date:** December 8, 2024