Report on ECC Python Script Functions

Overview

This script facilitates various operations related to Elliptic Curve Cryptography (ECC). It provides functions to validate curves, check points on curves, add and multiply points, and implement ECC-based cryptographic protocols like Diffie-Hellman and encryption/decryption.

inv_mod(k, n)

Computes the modular inverse of \(k \) modulo \(n \) using the Extended Euclidean Algorithm.

Parameters: k, n

Output: inverse of k

```
def inv_mod(k, n):
    # Return inverse modulo if n is not None, otherwise float
division
    if n is None:
        return 1 / k
    # Extended Euclidean Algorithm to find modular inverse
    if k == 0:
        raise ZeroDivisionError("division by zero")
    if k < 0:
        return n - inv_mod(-k, n)
    s, old_s = 0, 1
    t, old_t = 1, 0
    r, old_r = n, k
    while r != 0:
        quotient = old_r // r
```

```
        old_r, r = r, old_r - quotient * r

        old_s, s = s, old_s - quotient * s

        old_t, t = t, old_t - quotient * t

    return old_s % n
```

valid_curve(a, b, n=None)

Determines if the curve defined by the equation \(y^2 = x^3 + ax + b \) is non-singular.

Parameters: a, b, n

Output: True or False

```
def valid_curve(a, b, n=None):

    discriminant = 4 * a**3 + 27 * b**2

    return discriminant != 0 if n is None else discriminant % n != 0
```

is_point_on_curve(x, y, a, b, n=None)

Checks if a given point \((x, y)\) lies on the specified elliptic curve.

Parameters: x, y, a, b, n

Output: True or False

```
def is_point_on_curve(x, y, a, b, n=None):

    left_side = y**2

    right_side = x**3 + a*x + b

    if n is not None:

        left_side %= n

        right_side %= n

    return left_side == right_side
```

ecc_add(p, q, a, b, n=None)

Adds two points \(p \) and \(q \) on the elliptic curve.

Parameters: p, q, a, b, n

Output: p+q

```
def ecc_add(p, q, a, b, n=None):
    if p == (0, 0):
        return q
    if q == (0, 0):
        return p
    if p[0] == q[0] and p[1] != q[1]:
        return (0, 0)  # p + -p = 0 (point at infinity)

    if p == q:
        if p[1] == 0:  # point at infinity
            return (0, 0)
        # Calculate the slope (lambda) for the tangent line at p
        num = (3 * p[0]**2 + a)
        denom = (2 * p[1])
    else:
        # Calculate the slope (lambda) for the line through p and q
        num = (q[1] - p[1])
        denom = (q[0] - p[0])

    if n:  # If modular arithmetic is needed
```

```
        denom = inv_mod(denom, n)  # Modular inverse of the
denominator
        lam = (num * denom) % n
    else:
        lam = num / denom

    # Calculate coordinates of the resulting point
    x3 = lam**2 - p[0] - q[0]
    y3 = lam * (p[0] - x3) - p[1]

    if n:
        x3 %= n
        y3 %= n

    return (x3, y3)
```

ecc_multiply(point, k, a, b, n=None)

Multiplies a point \(p \) by an integer \(k \).

Parameters: point, k, a, b

Output: k*point

```
def ecc_multiply(point, k, a, b, n=None):
    result = (0, 0)  # Start with the identity element (point at
infinity)
    addend = point
    while k:
```

```
        if k & 1:
            result = ecc_add(result, addend, a, b, n)
        addend = ecc_add(addend, addend, a, b, n)
        k >>= 1

    return result
```

ecc_diffie_hellman(a, b, n=None)

Implements the ECC-based Diffie-Hellman protocol to demonstrate secure key exchange.

Parameters: a, b, n

```
def ecc_diffie_hellman(a, b, n=None):
    G = (get_input("Enter generator point Gx: "), get_input("Enter generator point Gy: "))
    if not is_point_on_curve(G[0], G[1], a, b, n):
        print("Generator point is not on the curve.")
        return
    private_A = get_input("Enter Alice's private key: ")
    private_B = get_input("Enter Bob's private key: ")
    public_A = ecc_multiply(G, private_A, a, b, n)
    public_B = ecc_multiply(G, private_B, a, b, n)
    shared_A = ecc_multiply(public_B, private_A, a, b, n)
    shared_B = ecc_multiply(public_A, private_B, a, b, n)
    print(f"Alice's public key: {public_A}")
    print(f"Bob's public key: {public_B}")
```

```
    if shared_A == shared_B:
        print(f"Shared secret: {shared_A}")
    else:
        print("Error calculating shared secret.")
```

ecc_encryption_decryption(a, b, n=None)

Demonstrates ECC-based encryption and decryption of a message.

Parameters: a, b, n

```
def ecc_encryption_decryption(a, b, n=None):
    public_key = (get_input("Enter public key Px: "),
get_input("Enter public key Py: "))
    if not is_point_on_curve(public_key[0], public_key[1], a, b, n):
        print("Public key is not on the curve.")
        return
    G = (get_input("Enter generator point Gx: "), get_input("Enter
generator point Gy: "))
    message = (get_input("Enter message point Mx: "),
get_input("Enter message point My: "))
    if not is_point_on_curve(message[0], message[1], a, b, n):
        print("Message point is not on the curve.")
        return
    k = get_input("Enter random integer k: ")


    # Encryption
    C1 = ecc_multiply(G, k, a, b, n)
```

```
    C2 = ecc_add(message, ecc_multiply(public_key, k, a, b, n), a,
b, n)

    print(f"Encrypted message: C1 = {C1}, C2 = {C2}")



    private_key = get_input("Enter Receiver's private key to
decrypt: ")

    # Decryption

    kP = ecc_multiply(C1, private_key, a, b, n)  # Calculate k*P
using private key

    inv_kP = (kP[0], -kP[1] % n if n else -kP[1])  # Inverse of kP

    decrypted_message = ecc_add(C2, inv_kP, a, b, n)

    print(f"Decrypted message: {decrypted_message}")
```

Conclusion

This script is a practical tool for exploring and implementing fundamental and advanced ECC operations, providing hands-on experience with real-world cryptographic applications.