Course Name:	Applied Cryptography	Semester:	V
Date of Performance:	_23__ / _09__ / __25____	DIV/ Batch No:	D-2
Student Name:	Shreyans Tatiya	Roll No:	16010123325

Experiment No: 6

Title: Implementation of ECC as discrete logarithm problem

Aim and Objective of the Experiment:

- Understand the Concept of ECC
- Implentation

COs to be achieved:

CO3: Analyze and Implement Symmetric and ASymmetric Key Cryptography Algorithms

Books/ Journals/ Websites referred:

1. https://www.vmware.com/topics/elliptic-curve-cryptography
2.

Theory:

Define discrete logarithm problem for elliptic curves.:

Consider the equation Q = kP where Q, P ∈ EP(a, b) and k < p.
It is relatively easy to calculate Q given k and P, but it is hard to determine k given Q and P. This is called the discrete logarithm problem for elliptic curves.

Example:
Consider the group $E_{23}(9,17)$.
This group is defined by the equation : $y^2 \bmod 23 = (x^3 + 9x + 17) \bmod 23$.

What is the discrete logarithm k of Q = (4, 5) to the base P = (16, 5)?

The brute-force method is to compute multiples of P until Q is found. Thus,
P = (16,5);
2P = (20, 20);
3P = (14, 14);
4P = (19, 20);
5P = (13, 10);
6P = (7, 3);
7P = (8, 7);

8P = (12, 17);
9P = (4, 5) =Q

Because 9P = (4, 5) = Q, the discrete logarithm Q = (4, 5) to the base P = (16, 5) is k = 9. In a real application, k would be so large as to make the Brute Force approach infeasible.

Problem statement

Part I

Implement point addition P+Q.

Implement point doubling 2P / Implement scalar multiplication kP using double-and-add method.

Part II - Given any points P(xp,yp) and Q(xq,yq), compute K, such that Q=K*P.

Given Q = K × P, attempt to compute K using brute force (feasible only with very small curves).

Show that as p grows larger, brute force becomes computationally infeasible.

Code :

Point Addition & Doubling

```
import time


# Modular inverse
def mod_inv(a, p):
    return pow(a, -1, p)

# Point addition
def point_add(P, Q, a, p):
    if P is None: return Q
    if Q is None: return P
    (x1, y1), (x2, y2) = P, Q
    if x1 == x2 and (y1 + y2) % p == 0:
```

```
            return None
        if P != Q:
            m = ((y2 - y1) * mod_inv(x2 - x1, p)) % p
        else:
            m = ((3 * x1**2 + a) * mod_inv(2 * y1, p)) % p
        x3 = (m*m - x1 - x2) % p
        y3 = (m*(x1 - x3) - y1) % p
        return (x3, y3)
```

Scalar Multiplication (Double-and-Add)

```
def scalar_mult(k, P, a, p):

    result = None
    addend = P
    while k:
        if k & 1:
            result = point_add(result, addend, a, p)
        addend = point_add(addend, addend, a, p)
        k >>= 1
    return result
```

Brute Force Discrete Logarithm

```
def discrete_log_bruteforce(P, Q, a, p, max_k=2000):

    R = None
    for k in range(1, max_k+1):
        R = point_add(R, P, a, p)
        if R == Q:
            return k
    return None
```

Main with timing experiments

```
if __name__ == "__main__":

    experiments = [
        # (p, a, b, P, Q, search_limit)
        (23, 9, 17, (16,5), (4,5), 100),      # Example from question
        (211, 2, 7, (2,2), None, 500),        # Q auto-generated
        (509, 3, 5, (1,2), None, 1000),
        (1009, 2, 3, (5,1), None, 2000),
    ]

    print(f"{'Prime':<8} {'a':<3} {'b':<3} {'P':<12} {'Q':<12} {'Found
k':<8} {'Time (s)':<10}")
```

```
    print("-"*65)

    for (p, a, b, P, Q, max_k) in experiments:
        # If Q not provided, generate Q = kP with random k (say 50)
        if Q is None:
            k_true = 50
            Q = scalar_mult(k_true, P, a, p)

        start = time.perf_counter()
        k_found = discrete_log_bruteforce(P, Q, a, p, max_k)
        end = time.perf_counter()

        runtime = end - start
        print(f"{p:<8} {a:<3} {b:<3} {str(P):<12} {str(Q):<12}
{str(k_found):<8} {runtime:.6f}")
```

Output:

```
Prime   a  b  P         Q         Found k  Time (s)
-----------------------------------------------------------
23      9  17 (16, 5)   (4, 5)    9        0.000026
211     2  7  (2, 2)    (58, 195) 50       0.000047
509     3  5  (1, 2)    (17, 376) 50       0.000059
1009    2  3  (5, 1)    (624, 656) 50      0.000056
```

Post Lab Subjective/Objective type Questions:

1. Compare RSA and ECC.

- **RSA:** Based on integer factorization, requires large key sizes (2048–4096 bits) for security.

- **ECC:** Based on elliptic curve discrete logarithm problem, provides same security with much smaller keys (e.g., 256-bit ECC ≈ 3072-bit RSA).

- **Result:** ECC is faster, requires less storage, and is widely used in modern systems (TLS, mobile, blockchain).

2. List various applications of ECC

- **Secure communications** → TLS/SSL, HTTPS.

- **Digital signatures** → ECDSA (used in Bitcoin, Ethereum).

- **Key exchange** → ECDH (used in Signal, WhatsApp).
- **IoT and mobile devices** → because of small keys and low power usage.
- **Blockchain & Cryptocurrencies** → wallet addresses and transaction signing.

3. Significance of discrete logarithm problem solution in ECDH cryptanalysis process.

- **ECDH (Elliptic Curve Diffie-Hellman)** security relies on the hardness of solving the discrete logarithm problem.
- If an attacker can **solve DLP efficiently**, they can compute private keys from public keys and break ECDH.
- Thus, DLP infeasibility ensures **confidentiality and security** in ECC-based key exchange.

Conclusion:

This experiment shows that ECC offers stronger security than RSA with smaller keys and faster performance. Its reliance on the hardness of the discrete logarithm problem ensures secure key exchange and makes it highly suitable for modern applications