🔐 Elliptic Curve Cryptography (ECC) — Encryption & Decryption Notes

✅ Key Concepts

- ECC is a public-key cryptography method based on the algebraic structure of **elliptic curves** over finite fields.
- **Equation of the elliptic curve:**

$$y^2 = x^3 + ax + b$$

🧮 Key Generation

- Let:
  - G: Generator point on the curve
  - n_A: Private key of User A
  - n_B: Private key of User B
  - $P_A = n_A \cdot G$: Public key of A
  - $P_B = n_B \cdot G$: Public key of B

📩 ECC Encryption

1. **Message** M is encoded as a point P_m on the curve.
2. Choose a **random integer** k.
3. Compute the **ciphertext** as a pair of points:

$$C_m = \{kG,\ P_m + kP_B\}$$

- First point: kG
- Second point: $P_m + kP_B$

🔓 ECC Decryption

User B receives $C_m = \{kG,\ P_m + kP_B\}$

1. Multiply first point with B's private key:

$$n_B \cdot kG = kP_B$$

2. Subtract kP_B from the second point:

$$(P_m + kP_B) - kP_B = P_m$$

Thus, the original message point P_m is recovered.

🔓 **ECC Decryption**

User B receives $C_m = \{kG,\ P_m + kP_B\}$

1. Multiply first point with B's private key:

$$n_B \cdot kG = kP_B$$

2. Subtract kP_B from the second point:

$$(P_m + kP_B) - kP_B = P_m$$

Thus, the original message point P_m is recovered.

🧠 **Efficiency Comparison (ECC vs RSA/DSA)**

Scheme	Bit Size (Security Level)	Equivalent RSA Bit Size
ECC	112 bits	512 bits
ECC	256 bits	3072 bits

- ECC provides **equivalent security** to RSA with **smaller keys**, resulting in:
  - Lower computational cost
  - Less memory and bandwidth usage

This is a simple SageMath code that demonstrates how to hide a message using encryption, and how the receiver can decrypt it to retrieve the original message.

```
# Define a small prime field
p = 211  # small prime number
F = GF(p)

# Define elliptic curve y^2 = x^3 + ax + b
a = 0
b = -4
E = EllipticCurve(F, [a, b])

# Choose a generator point G on the curve
G = E.random_point()
while G.order() < 20:
   G = E.random_point()

print("Generator G:", G)

# Key generation
na = 15                  # Private key of Alice
```

```
nb = 25                # Private key of Bob

PA = na * G            # Alice's public key
PB = nb * G            # Bob's public key

print("Alice's Public Key PA:", PA)
print("Bob's Public Key PB:", PB)

# Message as a point on the curve (simulate by picking random point)
Pm = E.random_point()
print("Original Message Point Pm:", Pm)

# Encryption (by Alice to Bob)
k = 19                 # random ephemeral key
C1 = k * G
C2 = Pm + k * PB

print("Ciphertext C1:", C1)
print("Ciphertext C2:", C2)

# Decryption (by Bob)
S = nb * C1            # shared secret
Pm_recovered = C2 - S

print("Decrypted Message Point:", Pm_recovered)

# Check
assert Pm == Pm_recovered, "Decryption failed!"
```

You can see the original message (16:111:1) and decrypted message are same (16:111:1).

```
Generator G: (29 : 139 : 1)
Alice's Public Key PA: (195 : 139 : 1)
Bob's Public Key PB: (5 : 200 : 1)
Original Message Point Pm: (16 : 111 : 1)
Ciphertext C1: (14 : 182 : 1)
Ciphertext C2: (160 : 8 : 1)
Decrypted Message Point: (16 : 111 : 1)
sage:
```