Algorithms:

1) K-means Algorithm :

Sr. No.	Original Image Size	Compressed Image Size
1	6,641 KB (6.48 MB)	995 KB
2	1,726 KB (1.68 MB)	269 KB
3	1,386 KB (1.35 MB)	266 KB
4	3,868 KB (3.77 MB)	869 KB
5	158 KB	33.8 KB
6	340 KB	94.1 KB

What is ChaCha20 Encryption?

- ChaCha20 is a symmetric encryption algorithm that uses a 256-bit key to encrypt and decrypt data.
- It was developed by Daniel J. Bernstein, a well-known cryptographer, in 2008 as a stream cipher, and later revised in 2014 as a block cipher.
- The ChaCha20 encryption algorithm is designed to provide both speed and security.
- It is designed to be resistant to known attacks, including differential cryptanalysis and linear cryptanalysis.
- Additionally, it is designed to be highly parallelizable, which means that it can be easily implemented on multi-core processors and other high-performance computing systems.

How does ChaCha20 Encryption work?

- ChaCha20 is a block cipher, which means that it encrypts data in fixed-size blocks.
- Each block of data is encrypted using a specific key, which is generated by the encryption algorithm.
- The key is used to generate a stream of pseudo-random bits, which are XORed with the plaintext to produce the ciphertext.

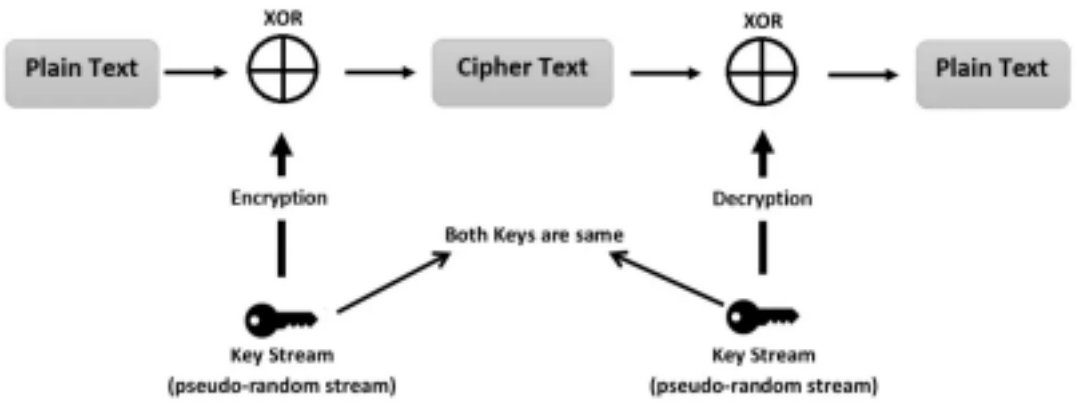
Here are the basic steps involved in the ChaCha20 algorithm:

Encryption:

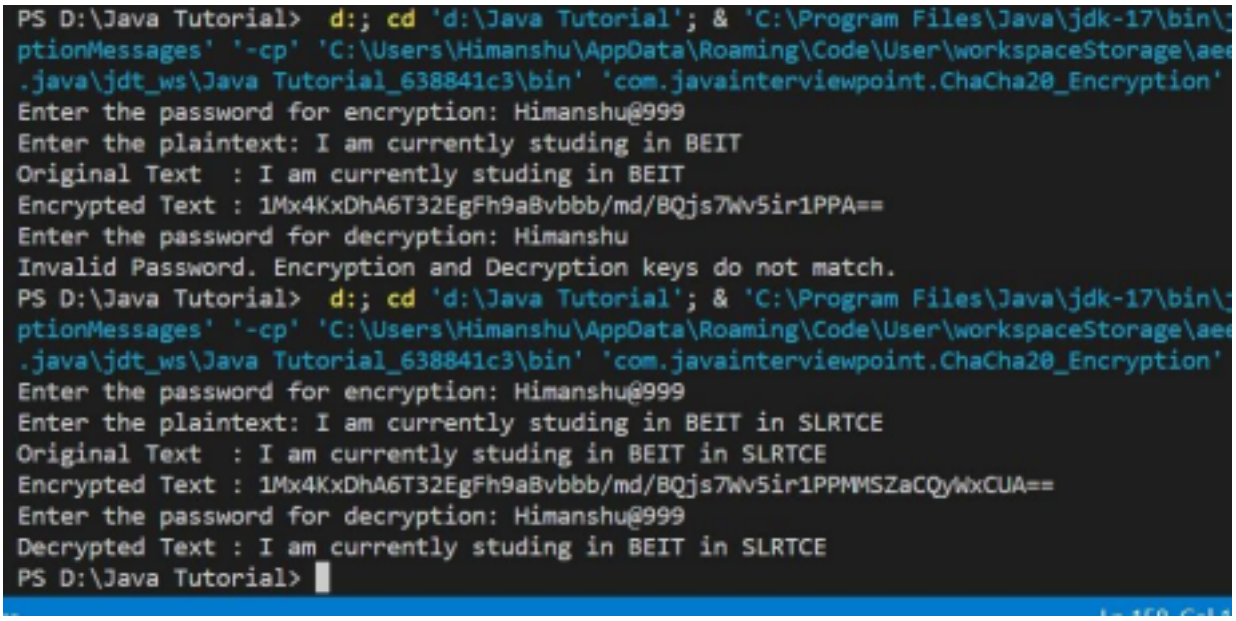
1. **Initialization**: Set up an initial state consisting of a 256-bit key, a 96-bit nonce (IV), and a 32-bit counter.
2. **Key Expansion**: Use the key and nonce to expand them into a series of 512-bit blocks (the ChaCha20 key schedule).
3. **Block Generation**: Generate a keystream by repeatedly applying a quarter-round function to the ChaCha20 state, typically 20 rounds.
4. **XOR Operation**: XOR the plaintext data with the keystream to produce the ciphertext.

Decryption:

1. **Initialization**: Use the same key, nonce (IV), and counter as used in encryption to set up the initial state.
2. **Key Expansion**: Perform the key expansion again to create the same keystream as used in encryption.
3. **Block Generation**: Generate the same keystream by applying the quarter-round function the same number of times (usually 20 rounds).
4. **XOR Operation**: XOR the ciphertext with the keystream to recover the plaintext.



Example:



What is ECC?

- ECC is a type of public-key cryptography.
- It relies on the mathematics of elliptic curves to provide strong security with relatively short key sizes compared to other public-key algorithms.

How ECC Works:

1. **Key Pair Generation:**
    - ECC involves the creation of a key pair:
        - **Private Key:** A secret number known only to the owner.
        - **Public Key:**A related point on the elliptic curve derived from the private key.
2. **Digital Signature Creation:**
    - To sign a message, the sender performs the following steps:
        - Hash the message to create a fixed-size message digest.
        - Use their private key to create a unique digital signature for the message digest.
3. **Digital Signature Verification:**

- The recipient of the message can verify the digital signature using the sender's public key:
    - Hash the received message to obtain a message digest.
    - Use the sender's public key to verify the digital signature.
    - If the signature is valid, the message is considered authentic and unaltered.

ECC Digital Signature Process:

1. Key Pair Generation:

- Himanshu generates his ECC key pair consisting of a private key (Himanshu's private key, denoted as 'a_H') and a corresponding public key (Himanshu's public key, denoted as 'Q_H').
- Aditya generates his ECC key pair consisting of a private key (Aditya's private key, denoted as 'a_A') and a corresponding public key (Aditya's public key, denoted as 'Q_A').

2. Digital Signature Creation (by Himanshu):

- Himanshu wants to sign a message (M) that he intends to send to Aditya: •
  He computes the message digest (H(M)).
  - Himanshu generates a random number (k_H).
  - Using his private key (a_H), Himanshu computes R_H = k_H * G, where G is the base point of the elliptic curve.
  - Himanshu computes s_H = (H(M) + a_H * R_H) / k_H.

3. Digital Signature Verification (by Aditya):

- Aditya receives M and the digital signature (R_H, s_H) from Himanshu.
- Aditya computes the message digest H(M).
- Aditya calculates the point P_H = s_H * G - H(M) * Q_H.
- If P_H is equal to R_H, then the signature is valid, and Aditya knows that the message is from Himanshu and hasn't been tampered with.

4. Message Authentication:

- Aditya is confident that the message received is indeed from Himanshu and hasn't been altered in transit because the digital signature verification was successful.

In this scenario, Himanshu uses his private key (a_H) to create a digital signature, and Aditya uses Himanshu's public key (Q_H) to verify the signature, ensuring the authenticity and

integrity of the message exchanged between them.

Working Image: