

<b>Name</b>	Prakriti Sharma
<b>Student ID</b>	A20575259
<b>Course ID</b>	CS458
<b>Title</b>	Coding Assignment II

## INDEX

Sr. No.	Content	Page No.
1.	Introduction to the assignment	2
2.	Details about the Algorithms, Modes & Ciphers Used	2
3.	Encryption Using Substitution Cipher (Shift & Permutation: About, Implementation, Dry Run and Conclusion)	3
4.	Encryption Using Transposition Cipher (Simple and Double Transposition: About, Implementation, Dry Run and Conclusion)	7
5.	Encryption Using Vigen 'ere Cipher (About, Implementation, Dry Run and Conclusion)	12
6.	Encryption Algorithms (AES, DES, 3DES: About, Implementation, Dry Run and Conclusion)	14
7.	Encryption Modes (ECB, CBC, CFB, OFB: About, Implementation, Dry Run and Conclusion)	18
8.	Decryption Using Substitution Cipher (Shift & Permutation: About, Implementation, Dry Run and Conclusion)	20
9.	Decryption Using Transposition Cipher (Simple and Double Transposition: About, Implementation, Dry Run and Conclusion)	24
10.	Decryption Using Vigen 'ere Cipher (Implementation, Dry Run and Conclusion)	27
11.	Decryption Algorithms (DES, 3DES: Implementation, Dry Run and Conclusion)	30
12.	Decryption Modes (ECB, CBC, CFB, OFB: Implementation, Dry Run and Conclusion)	31
13.	Screenshots of working code and all Ciphers/Algorithms/Modes for Encryption and Decryption(both)	37
14.	Pasted Code for reference	44

**Introduction to the assignment:**

The code has been implemented in JAVA language. The basic functionalities covered are as follows.

This assignment focusses on the Encryption Decryption methods of various types listed below:

- I. Substitution cipher
  - ✓ Shift Cipher
  - ✓ Permutation Cipher
- II. Transposition ciphers
  - ✓ Simple Transposition
  - ✓ Double Transposition
- III. Vigen 'ere Cipher
- IV. Different encryption algorithms
  - ✓ AES-128
  - ✓ DES
  - ✓ 3DES
- V. Different encryption modes
  - ✓ ECB
  - ✓ CBC
  - ✓ CFB
  - ✓ OFB

## **Shift Cipher:**

### ***About:***

The Shift Cipher, commonly known as the Caesar Cipher, is one of the oldest and simplest forms of encryption. It operates by shifting the letters of the alphabet by a fixed number of positions, defined by a key.

- How It Works: For example, if the key is 3, each letter in the plaintext will be shifted three places down the alphabet. Thus:

- 'A' becomes 'D'
- 'B' becomes 'E'
- 'C' becomes 'F'
- ...
- 'X' becomes 'A'
- 'Y' becomes 'B'
- 'Z' becomes 'C'

- Key Characteristics:
  - Substitution Method: Each letter in the plaintext is substituted with another letter that is a fixed number of places down the alphabet.
  - Wrap-around: When the end of the alphabet is reached, the shifting wraps around to the beginning.
  - Non-letter Preservation: Non-alphabetic characters (e.g., spaces, punctuation, numbers) remain unchanged.
  - Simple Implementation: Due to its straightforward nature, it is easy to implement and understand.

### ***Line-by-Line Explanation of the Code (Implementation):***

1. Method Declaration: The method is defined to take a string of text and an integer shift value.
2. StringBuilder Initialization: A StringBuilder object is created to build the ciphertext, as it allows for efficient appending of characters.
3. Loop Through Each Character: A loop is initiated to iterate through each character of the input text.
4. Character Assignment: Each character from the text is assigned to a variable for processing.
5. Check for Letters: A conditional statement checks if the character is a letter (either uppercase or lowercase).
6. Base Character Determination: Depending on whether the character is uppercase or lowercase, a base character ('A' or 'a') is assigned.
7. Shift Calculation: The character is shifted using the formula that involves its position relative to the base, applying the shift value, and wrapping around the alphabet using modulo arithmetic.
8. Append Shifted Character: The newly shifted character is appended to the StringBuilder to form the ciphertext.
9. Return Ciphertext: Once all characters have been processed, the method returns the encrypted string.

### ***Dry Run of the Code:***

Let's conduct a dry run of the Shift Cipher using an example to illustrate how the algorithm operates.

Example Input:

- Plaintext: "HELLO WORLD"
- Shift: 3

Dry Run Steps:

1. Initialization:
  - The input text is "HELLO WORLD".
  - The shift value is 3.
  - An empty StringBuilder for ciphertext is created.
2. Iterating Through Each Character:
  - Character 1 ('H'):
    - It is a letter. The base is 'A'.
    - The calculation:  $(\text{'H'} - \text{'A'} + 3) \% 26 + \text{'A'}$  results in 'K'.
    - Append 'K' to the ciphertext.
  - Character 2 ('E'):
    - It is a letter. The base is 'A'.
    - The calculation:  $(\text{'E'} - \text{'A'} + 3) \% 26 + \text{'A'}$  results in 'H'.
    - Append 'H' to the ciphertext.
  - Character 3 ('L'):
    - It is a letter. The base is 'A'.
    - The calculation:  $(\text{'L'} - \text{'A'} + 3) \% 26 + \text{'A'}$  results in 'O'.
    - Append 'O' to the ciphertext.
  - Character 4 ('L'):
    - It is a letter. The same calculation results in 'O'.
    - Append 'O' to the ciphertext.
  - Character 5 ('O'):
    - It is a letter. The calculation:  $(\text{'O'} - \text{'A'} + 3) \% 26 + \text{'A'}$  results in 'R'.
    - Append 'R' to the ciphertext.
  - Character 6 (space):
    - It is not a letter, so append it unchanged to the ciphertext.
  - Character 7 ('W'):
    - It is a letter. The base is 'A'.
    - The calculation:  $(\text{'W'} - \text{'A'} + 3) \% 26 + \text{'A'}$  results in 'Z'.
    - Append 'Z' to the ciphertext.
  - Character 8 ('O'):
    - It is a letter. The calculation results in 'R'.
    - Append 'R' to the ciphertext.

- Character 9 ('R'):
    - It is a letter. The calculation results in 'U'.
    - Append 'U' to the ciphertext.
  - Character 10 ('L'):
    - It is a letter. The calculation results in 'O'.
    - Append 'O' to the ciphertext.
  - Character 11 ('D'):
    - It is a letter. The calculation results in 'G'.
    - Append 'G' to the ciphertext.
3. Final Ciphertext:
- The final encrypted text is "KHOOR ZRUOG".

### ***Conclusion:***

The Shift Cipher is a simple yet effective way to encrypt messages by shifting letters of the alphabet by a defined key. The provided code implements this technique, preserving non-letter characters and efficiently building the resulting ciphertext. The dry run demonstrates how each character is processed to produce the final encrypted output.

### **Permutation Cipher:**

#### ***About:***

The Permutation Cipher is a method of encryption that rearranges the characters of the plaintext according to a specific key. This technique does not involve substituting characters but rather changing their positions based on a defined sequence, effectively creating a new order for the characters.

- How It Works:
  - A key is defined as an array of integers, where each integer represents the position of the character in the plaintext to be rearranged. For example, a key [3, 1, 4, 2] indicates that the first character of the plaintext will be moved to the third position, the second character to the first position, the third character to the fourth position, and the fourth character to the second position.
  - The plaintext is divided into blocks that match the length of the key. If the length of the plaintext is not a multiple of the block size, it is padded with spaces or another character to ensure all blocks are the same size.
- Key Characteristics:
  - Block-based Operation: The plaintext is processed in fixed-size blocks, allowing for a systematic approach to rearranging characters.
  - Rearrangement: The permutation does not change the characters themselves but changes their order, making it a simple yet effective form of encryption.
  - Padding: Non-full blocks are padded to maintain uniformity in processing.

### ***Line-by-Line Explanation of the Code (Implementation):***

1. Method Declaration: The method `permutationEncryptMethod` is defined to take a string plaintext and an integer array key.
2. Initialization: A `StringBuilder` named `ciphertext` is initialized to build the resulting encrypted string.
3. Block Size Calculation: The block size is determined by the length of the provided key.

4. **Padding the Plaintext:** A `StringBuilder` is used to create a mutable version of the plaintext. If the length of the plaintext is not a multiple of the block size, spaces are appended until it is.
5. **Processing Each Block:** A loop iterates through the plaintext in increments of the block size.
6. **Block Character Array:** A character array named `block` is created to hold the characters of the current block.
7. **Rearranging Characters:**
  - A nested loop iterates through the block size, rearranging the characters according to the permutation key.
  - The position of each character in the block is determined by the key. The expression `key[j] - 1` is used because arrays are zero-indexed.
8. **Appending the Rearranged Block:** The rearranged block is converted back to a string and appended to the ciphertext.
9. **Return Ciphertext:** Finally, the method returns the fully constructed ciphertext.

### ***Dry Run of the Code :***

Let's conduct a dry run of the Permutation Cipher using an example to illustrate how the algorithm operates.

Example Input:

- Plaintext: "HELLOWORLD"
- Key: [3, 1, 4, 2]

Dry Run Steps:

1. **Initialization:**
  - The input plaintext is "HELLOWORLD".
  - The key is [3, 1, 4, 2].
  - An empty `StringBuilder` for ciphertext is created.
  - The block size is calculated to be 4 (the length of the key).
2. **Padding the Plaintext:**
  - The length of "HELLOWORLD" is 10, which is not a multiple of 4.
  - Spaces are appended until the length is a multiple of 4.
  - The padded plaintext becomes "HELLOWORLD ".
3. **Processing Each Block:**
  - The loop iterates through the plaintext in blocks of 4 characters:
    - First Block (from index 0 to 3: "HELL"):
      - An empty block array of size 4 is created.
      - Rearranging:

- `key[0]` (3): Take 'L' (index 2).
        - `key[1]` (1): Take 'H' (index 0).
        - `key[2]` (4): Take 'L' (index 3).
        - `key[3]` (2): Take 'E' (index 1).

- The rearranged block is "LHEL".
- Append "LHEL" to the ciphertext.
- Second Block (from index 4 to 7: "OWOR"):
  - Rearranging:
 

- key[0] (3): Take 'O' (index 6).
    - key[1] (1): Take 'W' (index 4).
    - key[2] (4): Take 'R' (index 7).
    - key[3] (2): Take 'O' (index 5).
  - The rearranged block is "OWRO".
  - Append "OWRO" to the ciphertext.
- Third Block (from index 8 to 11: "LD "):
  - Rearranging:
 

- key[0] (3): Take ' ' (index 10).
    - key[1] (1): Take 'L' (index 8).
    - key[2] (4): Take 'D' (index 9).
    - key[3] (2): Take ' ' (index 11).
  - The rearranged block is "L D ".
  - Append "L D " to the ciphertext.

#### 4. Final Ciphertext:

- The final encrypted text is "LHEL OWRO L D ".

### ***Conclusion:***

The Permutation Cipher rearranges the characters of the plaintext based on a specified key. The provided code implements this technique, efficiently padding the plaintext as necessary and processing it in blocks to create the ciphertext. The dry run illustrates how each block of characters is transformed according to the key, resulting in the final encrypted output.

### **Simple Transposition Cipher:**

#### ***About:***

The Simple Transposition Cipher is a method of encryption that rearranges the characters of the plaintext based on a defined key, similar to the Permutation Cipher but structured in a grid format. This technique changes the positions of the characters without altering the characters themselves, making it a straightforward yet effective form of encryption.

- How It Works:
  - The plaintext is arranged in a grid with a number of columns equal to the length of the key. The characters are filled into the grid column by column.
  - The ciphertext is generated by reading the grid according to the order specified by the key, which determines the column arrangement for encryption.
- Key Characteristics:
  - Grid Structure: The plaintext is organized into a grid (2D array), facilitating a structured approach to rearranging characters.

- Column-based Reading: The ciphertext is generated by reading the grid based on the provided key.
- Padding: If the plaintext is shorter than the grid size, spaces are added to fill the remaining cells.

### *Line-by-Line Explanation of the Code (Implementation):*

1. **Method Declaration:** The method `simpleTranspositionEncryptMethod` is defined to take a string plaintext and an integer array key.
2. **Initialization:** A `StringBuilder` named `ciphertext` is initialized to construct the resulting encrypted string.
3. **Row Calculation:** The number of rows for the grid is calculated based on the length of the plaintext divided by the length of the key. `Math.ceil` ensures that any remainder results in an additional row.
4. **Grid Creation:** A 2D character array (grid) is initialized with dimensions based on the number of rows and the length of the key.
5. **Filling the Grid:**
  - A loop iterates through the grid, filling it with characters from the plaintext.
  - If the index exceeds the length of the plaintext, the remaining cells in the grid are padded with spaces.
6. **Rearranging the Grid:**
  - A loop iterates over the key. Each value in the key determines the order in which columns are read to create the ciphertext.
  - For each key value, the corresponding column index (`keyCol`) is calculated by subtracting 1 (to adjust for zero-based indexing).
  - A nested loop reads characters from the specified column and appends them to the ciphertext.
7. **Return Ciphertext:** Finally, the method returns the fully constructed ciphertext.

### *Dry Run of the Code:*

Let's conduct a dry run of the Simple Transposition Cipher using an example to illustrate how the algorithm operates.

Example Input:

- Plaintext: "HELLO WORLD"
- Key: [3, 1, 4, 2]

Dry Run Steps:

1. **Initialization:**
  - The input plaintext is "HELLO WORLD".
  - The key is [3, 1, 4, 2].
  - An empty `StringBuilder` for ciphertext is created.
2. **Row Calculation:**
  - The length of the plaintext is 11.
  - The length of the key is 4.
  - Number of rows =  $\text{Math.ceil}(11 / 4) = 3$ .
3. **Grid Creation:**



- A grid of size 3x4 (3 rows, 4 columns) is created:

```
[ ['H', 'E', 'L', 'L'],
  ['O', '', 'W', 'O'],
  ['R', 'L', 'D', '']]
]
```

#### 4. Filling the Grid:

- The grid is filled with characters from the plaintext. After processing, the grid looks like this:

```
[ ['H', 'E', 'L', 'L'],
  ['O', '', 'W', 'O'],
  ['R', 'L', 'D', '']]
]
```

#### 5. Rearranging the Grid:

- Using the key [3, 1, 4, 2], we rearrange and read the grid:

- For key value 3 (column index 2):

- Read characters: 'L' (row 0), 'W' (row 1), 'D' (row 2).
- Append to ciphertext: "LWD".

- For key value 1 (column index 0):

- Read characters: 'H' (row 0), 'O' (row 1), 'R' (row 2).
- Append to ciphertext: "HOD".

- For key value 4 (column index 3):

- Read characters: 'L' (row 0), 'O' (row 1), '' (row 2).
- Append to ciphertext: "LO ".

- For key value 2 (column index 1):

- Read characters: 'E' (row 0), '' (row 1), 'L' (row 2).
- Append to ciphertext: "E L".

#### 6. Final Ciphertext:

- The final encrypted text is "LWDHODLO E L".

### ***Conclusion:***

The Simple Transposition Cipher rearranges the characters of the plaintext based on a specified key using a grid structure. The provided code implements this technique by filling a 2D array with characters from the plaintext and reading the array according to the order defined by the key. The dry run illustrates how the algorithm processes the input to produce the final encrypted output.

### ***Double Transposition Cipher:***

The Double Transposition Cipher enhances the security of the Simple Transposition Cipher by applying two rounds of transposition using two different keys. This method increases the complexity of the encryption, making it more resistant to attacks compared to a single transposition.

- How It Works:

1. First Transposition: The plaintext is first encrypted using a specified key, rearranging the characters in a grid format.
  2. Second Transposition: The result from the first transposition is then encrypted again using a second key, further scrambling the characters.
- Key Characteristics:
    - Increased Security: By applying two keys, the cipher makes it significantly harder for an attacker to decipher the plaintext without knowing both keys.
    - Two Layers of Rearrangement: Each key leads to a unique arrangement of the characters, contributing to a more complex final ciphertext.

### *Line-by-Line Explanation of the Code (Implementation):*

1. Method Declaration: The method `doubleTranspositionEncryptMethod` is defined to take a string plaintext and two integer arrays `key1` and `key2`.
2. First Transposition:
  - The first transposition is performed by calling `simpleTranspositionEncryptMethod` with the plaintext and `key1`.
  - The result of this first encryption is stored in the variable `firstPass`.
3. Second Transposition:
  - The method returns the result of the second transposition by calling `simpleTranspositionEncryptMethod` again, this time using `firstPass` and `key2`.

### *Dry Run of the Code:*

Let's conduct a dry run of the Double Transposition Cipher using an example to illustrate how the algorithm operates.

Example Input:

- Plaintext: "HELLO WORLD"

- Key1: [3, 1, 4, 2]
- Key2: [1, 4, 2, 3]

Dry Run Steps:

1. Initialization:
  - The input plaintext is "HELLO WORLD".
  - The first key is [3, 1, 4, 2], and the second key is [1, 4, 2, 3].
2. First Transposition (Using Key1):
  - The method `simpleTranspositionEncryptMethod` is called with the plaintext "HELLO WORLD" and `key1`.
  - Grid Creation:
    - Calculate the number of rows:  $\text{Math.ceil}(11 / 4) = 3$ .
    - Create a 3x4 grid:

```
[ [ 'H', 'E', 'L', 'L' ],
  [ 'O', ' ', 'W', 'O' ],
  [ 'R', 'L', 'D', ' ' ]]
```

]

- Rearranging the Grid (based on Key1):

- Using key [3, 1, 4, 2]:

- For key value 3 (column index 2): 'L', 'W', 'D' → "LWD".
- For key value 1 (column index 0): 'H', 'O', 'R' → "HOD".
- For key value 4 (column index 3): 'L', 'O', ' ' → "LO ".
- For key value 2 (column index 1): 'E', ' ', 'L' → "E L".

- The result from the first pass is:

- FirstPass: "LWDHODLO E L".

### 3. Second Transposition (Using Key2):

- The method `simpleTranspositionEncryptMethod` is called with `firstPass` ("LWDHODLO E L") and `key2`.

- Grid Creation:

- The length of `firstPass` is 13.
- Calculate the number of rows:  $\text{Math.ceil}(13 / 4) = 4$ .
- Create a 4x4 grid (note the additional space):

[ [ 'L', 'W', 'D', 'H' ],

[ 'O', 'D', 'L', ' ' ],

[ 'E', ' ', ' ', ' ' ],

[ ' ', ' ', ' ', ' ' ]

]

- Rearranging the Grid (based on Key2):

- Using key [1, 4, 2, 3]:

- For key value 1 (column index 0): 'L', 'O', 'E', ' ' → "LOE ".
- For key value 4 (column index 3): 'H', ' ', ' ', ' ' → "H ".
- For key value 2 (column index 1): 'W', 'D', ' ', ' ' → "WD".
- For key value 3 (column index 2): 'D', 'L', ' ', ' ' → "DL".

### 4. Final Ciphertext:

- Concatenating the results from the second pass:

- The final encrypted text is "LOE HWDL".

## Conclusion:

The Double Transposition Cipher enhances the Simple Transposition Cipher's security by applying two rounds of transposition using two distinct keys. The provided code effectively implements this technique by performing the first and second transpositions in sequence. The dry run illustrates the steps taken to transform the plaintext into a complex final ciphertext through two levels of rearrangement.

### **Vigen'ere Cipher:**

The Vigenère Cipher is a method of encrypting alphabetic text using a simple form of polyalphabetic substitution. It employs a keyword to dictate how much to shift each letter in the plaintext based on the corresponding letter of the keyword.

- How It Works:
  1. Keyword Preparation: The keyword is repeated or truncated to match the length of the plaintext. Each letter of the keyword represents a shift in the alphabet.
  2. Encryption Process: Each letter in the plaintext is shifted based on the corresponding letter in the keyword:
    - The position of each letter in the alphabet (A=0, B=1, ..., Z=25) is used to calculate the resulting letter in the ciphertext.
- Key Characteristics:
  - Polyalphabetic Cipher: Unlike monoalphabetic ciphers, the Vigenère Cipher uses multiple shifts depending on the keyword, making it more secure against frequency analysis.
  - Non-Letter Handling: Non-alphabetic characters are retained in the ciphertext without alteration.

### **Line-by-Line Explanation of the Code (Implementation):**

1. Method Declaration: The method `vigenereEncryptMethod` is defined to take two parameters: `plaintext` (the text to be encrypted) and `keyword` (the keyword used for encryption).
2. Keyword Normalization:
  - The keyword is converted to uppercase to ensure consistent handling of characters.
3. Encryption Loop:
  - A for loop iterates through each character in the plaintext.
4. Character Processing:
  - For each character, it checks if the character is a letter using `Character.isLetter(plainChar)`:
    - If it is a letter:
      - Calculate the shift by converting both the plaintext character and the corresponding keyword character to their positions in the alphabet (A=0, B=1, ..., Z=25):
        - `shift = (plainChar - 'A' + keyword.charAt(i % keyword.length()) - 'A') % 26;`
        - Here, `i % keyword.length()` allows the keyword to wrap around if it's shorter than the plaintext.
      - Append the corresponding ciphertext character, calculated by adding the shift back to 'A':
        - `ciphertext.append((char) (shift + 'A'));`
    - If it is not a letter:
      - Append the non-letter character unchanged to the ciphertext.
5. Return Statement: The method returns the final ciphertext as a string.

### **Dry Run of the Code:**

Let's conduct a dry run of the Vigenère Cipher using an example to illustrate how the algorithm operates.

Example Input:

- Plaintext: "ATTACKATDAWN"
- Keyword: "LEMON"

Dry Run Steps:

1. Initialization:

- The input plaintext is "ATTACKATDAWN".
- The keyword is "LEMON", which is converted to uppercase: "LEMON".

2. Keyword Adjustment:

- Repeat the keyword to match the length of the plaintext:
- Plaintext: A T T A C K A T D A W N
- Keyword: L E M O N L E M O N L E

3. Encryption Loop:

- Initialize an empty StringBuilder for ciphertext.
- Character by Character Processing:

▪ For A (Index 0):

▪ Shift:  $(0+11)\bmod 26=11(0+11)\bmod 26=11(0+11)\bmod 26=11 \rightarrow L$

▪ For T (Index 1):

▪ Shift:  $(19+4)\bmod 26=23(19+4)\bmod 26=23(19+4)\bmod 26=23 \rightarrow X$

▪ For T (Index 2):

▪ Shift:  $(19+12)\bmod 26=5(19+12)\bmod 26=5(19+12)\bmod 26=5 \rightarrow F$

▪ For A (Index 3):

▪ Shift:  $(0+14)\bmod 26=14(0+14)\bmod 26=14(0+14)\bmod 26=14 \rightarrow O$

▪ For C (Index 4):

▪ Shift:  $(2+13)\bmod 26=15(2+13)\bmod 26=15(2+13)\bmod 26=15 \rightarrow P$

▪ For K (Index 5):

▪ Shift:  $(10+11)\bmod 26=21(10+11)\bmod 26=21(10+11)\bmod 26=21 \rightarrow V$

▪ For A (Index 6):

▪ Shift:  $(0+4)\bmod 26=4(0+4)\bmod 26=4(0+4)\bmod 26=4 \rightarrow E$

▪ For T (Index 7):

▪ Shift:  $(19+12)\bmod 26=5(19+12)\bmod 26=5(19+12)\bmod 26=5 \rightarrow F$

▪ For D (Index 8):

▪ Shift:  $(3+14)\bmod 26=17(3+14)\bmod 26=17(3+14)\bmod 26=17 \rightarrow R$

▪ For A (Index 9):

▪ Shift:  $(0+13)\bmod 26=13(0+13)\bmod 26=13(0+13)\bmod 26=13 \rightarrow N$

- For W (Index 10):

▪ Shift:  $(22+11) \bmod 26 = 7(22 + 11) \bmod 26 = 7(22+11) \bmod 26 = 7 \rightarrow H$

- For N (Index 11):

▪ Shift:  $(13+4) \bmod 26 = 17(13 + 4) \bmod 26 = 17(13+4) \bmod 26 = 17 \rightarrow R$

#### 4. Final Ciphertext:

- The resulting ciphertext after processing all characters:
- Ciphertext: "LXFOPVEFRNHR"

### ***Conclusion:***

The Vigenère Cipher effectively encrypts text by utilizing a keyword to dictate letter shifts, resulting in a more secure encryption method compared to simple substitution ciphers. The provided code accurately implements this algorithm, ensuring that non-letter characters are preserved. The dry run demonstrates how each letter of the plaintext is transformed according to the corresponding letter of the keyword, resulting in the final ciphertext.

### **Encryption Algorithms:**

#### **AES Algorithm:**

##### ***About:***

Advanced Encryption Standard (AES) is a symmetric encryption algorithm widely adopted for securing data. It operates on fixed-size blocks (128 bits or 16 bytes) and supports key sizes of 128, 192, and 256 bits. AES is known for its speed and security and is used in various applications, including secure communications, file encryption, and data protection.

AES can operate in several modes, with Electronic Codebook (ECB) being one of the simplest. In ECB mode, each block of plaintext is encrypted independently, which can introduce vulnerabilities if identical blocks are present in the plaintext.

##### ***Line-by-Line Explanation of the Code (Implementation):***

The following section outlines the key components of the AES encryption method, detailing each part of the code:

1. Method Declaration:
  - The method is named `encryptAESMethod` and takes two parameters: `plaintext` (the text to be encrypted) and `key` (the secret key used for encryption). It throws an exception to handle any encryption-related errors.
2. Cipher Initialization:
  - The `Cipher` class is used to perform encryption operations. `Cipher.getInstance("AES/ECB/PKCS5Padding")` creates a `Cipher` object configured to use the AES algorithm in ECB mode with PKCS5 padding. Padding is necessary because AES requires the plaintext to be a multiple of the block size (16 bytes).
3. Cipher Configuration:
  - The line `cipher.init(Cipher.ENCRYPT_MODE, key)` initializes the cipher instance for encryption. The provided secret key is used to set up the internal state for the encryption process. This key must be the same during decryption to retrieve the original plaintext.
4. Encryption Process:
  - The `doFinal(plaintext.getBytes())` method is called on the cipher object. This converts the input plaintext into a byte array using the default character encoding and processes it through the cipher, encrypting it based on the initialized settings. The method returns an array of bytes representing the encrypted data.

#### 5. Base64 Encoding:

- The encrypted byte array is converted into a Base64 encoded string using `Base64.getEncoder().encodeToString(encryptedBytes)`. Base64 encoding transforms binary data into a text representation, which is easier to store and transmit over text-based protocols.

### *Dry Run of the code:*

Let's perform a dry run of the AES encryption process with a simple example.

Example Input:

- Plaintext: "HELLO WORLD"
- Key: A randomly generated 128-bit key, for instance, 3D5D3B550A68D5E8C4A004154F80E91A.

Step-by-Step Process:

1. Initialize Cipher:
  - The cipher is initialized with AES/ECB/PKCS5Padding settings. The key is prepared and provided for encryption.
2. Pad Plaintext:
  - The plaintext "HELLO WORLD" is padded to match the block size (16 bytes). In this case, it may become "HELLO WORLD " (with 3 spaces added).
3. Encrypt Each Block:
  - The padded plaintext is processed in 16-byte blocks. The first block "HELLO WORLD " is encrypted using the AES algorithm and the provided key, resulting in an encrypted byte array.
4. Output Encrypted Bytes:
  - The encrypted bytes are generated and converted into a Base64 encoded string.

Final Output:

- The result will be a Base64 encoded string representing the encrypted data, for example, "U2Fs dGVkX1+...".

### *Conclusion:*

The AES encryption method provides a secure way to encrypt data using a symmetric key algorithm. Each step in the process, from initializing the cipher to generating the encrypted output, is crucial for ensuring the confidentiality of the plaintext. The ECB mode, while straightforward, has its limitations, and care must be taken when using it in practice.

### **DES Algorithm:**

#### *About:*

Data Encryption Standard (DES) is a symmetric-key block cipher that was widely used for data encryption. It operates on 64-bit blocks of data and uses a 56-bit key for encryption and decryption. Although DES was a significant advancement in cryptography, it is now considered insecure due to its relatively short key length, which makes it vulnerable to brute-force attacks. As a result, it has largely been replaced by more secure algorithms like AES.

In the context of this implementation, DES is used in Electronic Codebook (ECB) mode with PKCS5 padding, similar to how AES is implemented.

### *Line-by-Line Explanation of the Code (Implementation):*

This section provides a detailed breakdown of the DES encryption method, explaining each part of the code line by line:

1. Method Declaration:
  - The method is named `encryptDESMethod` and accepts two parameters: `plaintext` (the text to be encrypted) and `key` (the secret key used for encryption). The method throws an exception to handle potential errors that may arise during encryption.
2. Cipher Initialization:
  - The line `Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding")` creates a `Cipher` object configured to use the DES algorithm in ECB mode with PKCS5 padding. This setting ensures that the plaintext is appropriately padded to match the block size required by DES.
3. Cipher Configuration:
  - The cipher is initialized for encryption using `cipher.init(Cipher.ENCRYPT_MODE, key)`. The provided secret key is necessary for the encryption process and must be the same for decryption to retrieve the original plaintext.
4. Encryption Process:
  - The plaintext is encrypted by calling `cipher.doFinal(plaintext.getBytes())`. This method converts the plaintext into a byte array using the default character encoding and processes it through the cipher, resulting in an array of encrypted bytes.
5. Base64 Encoding:
  - The encrypted byte array is converted to a Base64 encoded string using `Base64.getEncoder().encodeToString(encryptedBytes)`. Base64 encoding transforms the binary data into a text format suitable for storage and transmission over text-based protocols.

### *Dry Run of the code:*

Let's perform a dry run of the DES encryption process with a simple example.

Example Input:

- Plaintext: "HELLO"
- Key: A randomly generated 56-bit DES key, for instance, 12345678.

Step-by-Step Process:

1. Initialize Cipher:
  - A cipher object is created with DES/ECB/PKCS5Padding settings, using the provided key for initialization.
2. Pad Plaintext:
  - The plaintext "HELLO" is padded to fit the block size (8 bytes for DES). In this case, it may become "HELLO " (with 3 spaces added).
3. Encrypt Each Block:
  - The padded plaintext is processed as an 8-byte block. The block "HELLO " is encrypted using the DES algorithm and the specified key, resulting in an encrypted byte array.
4. Output Encrypted Bytes:
  - The encrypted bytes are then converted into a Base64 encoded string.

Final Output:

- The result will be a Base64 encoded string representing the encrypted data, for example, "FIiY9t12Dd8=".



### ***Conclusion:***

The DES encryption method provides a means of securing data using a symmetric key algorithm, similar to AES but with a shorter key length and block size. Each step, from cipher initialization to generating the encrypted output, is essential for maintaining the confidentiality of the plaintext. Although DES played a critical role in the development of cryptographic methods, it is now considered outdated and insecure, making it essential to use stronger alternatives in modern applications.

### **3DES Algorithm:**

#### ***About:***

3DES (Triple Data Encryption Standard) is a symmetric-key block cipher that applies the DES cipher algorithm three times to each data block. It was designed to provide a more secure encryption method than standard DES by effectively increasing the key length. While it was widely used in the past, it has been largely replaced by more secure algorithms such as AES due to vulnerabilities in its design and slower performance.

#### ***Line-by-Line Explanation of the Code (Implementation):***

1. Cipher Initialization:

- The line `Cipher cipher = Cipher.getInstance("DESede/ECB/PKCS5Padding");` creates a new Cipher object that uses the 3DES encryption algorithm in Electronic Codebook (ECB) mode with PKCS5 padding.
- Key Points:
  - "DESede" indicates that the Triple DES algorithm will be used.
  - "ECB" mode encrypts each block of data independently, which can lead to security issues in certain contexts.
  - "PKCS5Padding" ensures that the input plaintext is padded to be a multiple of the block size (8 bytes for DES).

2. Cipher Initialization for Encryption:

- The line `cipher.init(Cipher.ENCRYPT_MODE, key);` initializes the cipher object for encryption mode using the provided secret key.
- Key Points:
  - The key must be a valid `SecretKey` object representing the encryption key.
  - The cipher must be initialized before performing any encryption.

3. Encrypting the Plaintext:

- The line `byte[] encryptedBytes = cipher.doFinal(plaintext.getBytes());` processes the input plaintext to produce the encrypted data.
- Key Points:
  - `plaintext.getBytes()` converts the plaintext string into a byte array using the default character encoding.
  - `cipher.doFinal()` performs the encryption and returns the encrypted data as a byte array.

4. Encoding Encrypted Data:

- The line `return Base64.getEncoder().encodeToString(encryptedBytes);` converts the encrypted byte array into a Base64 encoded string.
- Key Points:

- Base64 encoding is used to represent binary data (the encrypted bytes) as an ASCII string, making it easier to transmit or store in a text format.

### ***Dry Run of the Code:***

Let's consider a dry run of the encrypt3DESMethod with the following example:

- Input:
  - plaintext: "HELLO WORLD"
  - key: A valid 3DES key (e.g., a 24-byte key).
- 1. Creating the Cipher Object:
  - The cipher is initialized with 3DES, ECB mode, and PKCS5 padding.
- 2. Initializing for Encryption:
  - The cipher is set to encryption mode using the provided key.
- 3. Encrypting the Plaintext:
  - The plaintext "HELLO WORLD" is converted to bytes: [72, 69, 76, 76, 79, 32, 87, 79, 82, 76, 68].
  - Since "HELLO WORLD" is not a multiple of 8 bytes, PKCS5 padding is applied, resulting in additional bytes (e.g., padding with spaces or a specific byte value) to make the total length a multiple of 8.
  - The padded byte array is then processed by the 3DES algorithm, producing the encrypted byte array (let's assume the output is [103, 28, 95, ...]).
- 4. Encoding the Encrypted Data:
  - The encrypted byte array is then converted to a Base64 encoded string, such as "gqW8w5..." for easy representation.
- 5. Return Value:
  - The method returns the Base64 encoded string representation of the encrypted data, which can be stored or transmitted securely.

This dry run outlines how the algorithm processes the input plaintext through initialization, encryption, and final encoding stages.

### ***Conclusion:***

While 3DES was once widely adopted for securing sensitive data, it is important to note that it is now considered less secure than modern encryption standards, such as AES. Therefore, for new applications and systems, it is advisable to use more robust encryption methods like AES, which offer better performance and security. Nevertheless, understanding 3DES provides valuable insights into the evolution of cryptographic techniques and the importance of strong encryption in protecting data confidentiality.

### **Encryption Modes:**

#### ***Overview of AES in Different Modes:***

The Advanced Encryption Standard (AES) is a widely used symmetric encryption algorithm that encrypts data in fixed-size blocks (128 bits). AES can operate in various modes, each with distinct characteristics that affect how data is encrypted and decrypted. Four commonly used modes of AES are Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), and Output Feedback (OFB). Below, we explain each mode with reference to the provided code.

##### ***1. AES in ECB Mode:***

**Description:**

In ECB mode, each block of plaintext is encrypted independently using the same key. This means that identical plaintext blocks will produce identical ciphertext blocks. While ECB is simple and fast, it is not recommended for encrypting sensitive data because it does not provide sufficient security. Patterns in the plaintext can be detected in the ciphertext, making it vulnerable to attacks.

**Code Explanation:**

- The method begins by obtaining an AES cipher instance configured for ECB mode.
- It initializes the cipher for encryption using the provided secret key.
- The plaintext is then converted to a byte array and encrypted using the doFinal method.
- Finally, the encrypted byte array is encoded to a Base64 string for easy storage or transmission.

**2. AES in CBC Mode:****Description:**

CBC mode improves upon ECB by chaining blocks together. The first block of plaintext is XORed with an Initialization Vector (IV) before encryption. Each subsequent block is XORed with the previous ciphertext block. This chaining process ensures that identical plaintext blocks will result in different ciphertext blocks, enhancing security.

**Code Explanation:**

- The method starts by acquiring a cipher instance configured for CBC mode.
- An IV is specified as an input parameter, and an IvParameterSpec object is created from it.
- The cipher is initialized for encryption with the key and IV.
- The plaintext is encrypted, and the resulting byte array is encoded in Base64 format for output.

**3. AES in CFB Mode:****Description:**

CFB mode allows encryption of data in smaller increments, making it suitable for streaming data. Similar to CBC, CFB uses an IV, but it encrypts the IV and XORs it with the plaintext. The result is that it can encrypt data of any size, and the encryption process does not require padding.

**Code Explanation:**

- The method begins by obtaining a cipher instance set to CFB mode.
- An IV is provided and encapsulated in an IvParameterSpec object.
- The cipher is initialized for encryption with the key and IV.
- The plaintext is then encrypted, and the encrypted bytes are returned as a Base64 string.

**4. AES in OFB Mode:****Description:**

OFB mode is similar to CFB but operates differently. Instead of using the ciphertext to encrypt the next block, it repeatedly encrypts the IV, creating a keystream. This keystream is XORed with the plaintext to produce ciphertext. OFB is advantageous in that it allows for parallel processing of the plaintext.

**Code Explanation:**

- The method starts by obtaining a cipher instance configured for OFB mode.
- An IV is provided and wrapped in an IvParameterSpec object.
- The cipher is initialized for encryption using the key and IV.
- The plaintext is encrypted, and the resulting byte array is encoded to Base64 format for easy handling.

### *Conclusion:*

Each AES mode has its unique way of processing plaintext to enhance security and adaptability based on the application's requirements. While ECB is the simplest and fastest, it lacks security, making CBC, CFB, and OFB more suitable for encrypting sensitive information. Understanding these modes allows developers to select the most appropriate one based on the context of use, balancing security, performance, and the nature of the data being encrypted.

### **Decryption Algorithms:**

#### **Shift Cipher Decryption:**

#### *Line-by-Line Explanation of the Code (Implementation):*

The provided code is a method designed to decrypt text that has been encrypted using the Shift Cipher. It takes in two parameters: the encrypted text and the shift value used during encryption. Below is a line-by-line explanation of the code:

1. Initialization:
  - A StringBuilder object named `decryptedText` is initialized to store the characters of the decrypted message as they are processed.
2. Iterating Through Characters:
  - A for-each loop iterates through each character of the `encryptedText`.
3. Handling Uppercase Letters:
  - Inside the loop, the first conditional checks if the character is an uppercase letter. If so:
    - The character is converted back to its original form by subtracting 'A' (to normalize it to 0), subtracting the shift value, and applying modulo 26 to handle wrap-around for letters.
    - The resulting value is converted back to a character by adding 'A' and appended to `decryptedText`.
4. Handling Lowercase Letters:
  - The next conditional checks if the character is a lowercase letter. The process is similar to that for uppercase letters:
    - The character is adjusted by subtracting 'a' instead of 'A' to maintain its position in the alphabet.
    - The resulting character is appended to `decryptedText`.
5. Handling Non-Letter Characters:
  - If the character is neither uppercase nor lowercase (e.g., spaces, punctuation), it is directly appended to `decryptedText` unchanged.
6. Returning the Result:
  - Finally, the method returns the decrypted text as a string.

### *Dry Run of the code:*

Let's consider a dry run using the following inputs:

- Encrypted Text: "Khoor Zruog"
  - Shift: 3
1. Initialization:

- `decryptedText` is initialized as an empty string.

2. Iteration:

- For the character 'K':
  - It is uppercase. Calculation:  $((75 - 65 - 3 + 26) \% 26) + 65 = 72$  (which corresponds to 'H').
  - `decryptedText` now contains "H".
- For the character 'h':
  - It is lowercase. Calculation:  $((104 - 97 - 3 + 26) \% 26) + 97 = 101$  (which corresponds to 'e').
  - `decryptedText` now contains "He".
- For the character 'o':
  - It is lowercase. Calculation:  $((111 - 97 - 3 + 26) \% 26) + 97 = 108$  (which corresponds to 'l').
  - `decryptedText` now contains "Hel".
- For the character 'o':
  - Similar to before, it results in 'l'.
  - `decryptedText` now contains "Hell".
- For the character 'r':
  - Calculation:  $((114 - 97 - 3 + 26) \% 26) + 97 = 111$  (which corresponds to 'o').
  - `decryptedText` now contains "Hello".
- For the space ' ':
  - It is a non-letter, so it is appended unchanged.
  - `decryptedText` now contains "Hello ".
- For the character 'Z':
  - Calculation:  $((90 - 65 - 3 + 26) \% 26) + 65 = 87$  (which corresponds to 'W').
  - `decryptedText` now contains "Hello W".
- For the character 'r':
  - Calculation:  $((114 - 97 - 3 + 26) \% 26) + 97 = 111$  (which corresponds to 'o').
  - `decryptedText` now contains "Hello Wo".
- For the character 'u':
  - Calculation:  $((117 - 97 - 3 + 26) \% 26) + 97 = 118$  (which corresponds to 'r').
  - `decryptedText` now contains "Hello Wor".
- For the character 'o':
  - Calculation:  $((111 - 97 - 3 + 26) \% 26) + 97 = 108$  (which corresponds to 'l').
  - `decryptedText` now contains "Hello Worl".
- For the character 'g':
  - Calculation:  $((103 - 97 - 3 + 26) \% 26) + 97 = 100$  (which corresponds to 'd').

- `decryptedText` now contains "Hello World".

### 3. Final Output:

- The method returns the decrypted text: "Hello World".

## ***Conclusion:***

The Shift Cipher, while straightforward, can easily be broken with frequency analysis, particularly with short shift values. However, it serves as an excellent introduction to classical cryptography concepts. The decryption method effectively reverses the shifting process used during encryption, demonstrating how symmetric key algorithms operate.

## **Permutation Cipher Decryption:**

### ***Line-by-Line Explanation of the Code (Implementation):***

The provided code is a method designed to decrypt text that has been encrypted using the Permutation Cipher. It takes two parameters: the ciphertext (the encrypted text) and a key (an array of integers representing the permutation). Below is a line-by-line explanation of the code:

#### 1. Initialization:

- A `StringBuilder` object named `plaintext` is initialized to store the characters of the decrypted message as they are processed.
- The `blockSize` variable is set to the length of the key, which determines how many characters are processed in each iteration.

#### 2. Processing Each Block:

- A `for` loop iterates through the ciphertext in increments of `blockSize`. This means the loop processes each block of characters from the ciphertext in groups of the size defined by the key.

#### 3. Creating a Block:

- For each iteration, a character array named `block` is created with the size of `blockSize` to hold the characters of the current block.

#### 4. Rearranging Characters:

- A nested `for` loop iterates over the indices of the block. For each index `j`, it uses the permutation key to place the corresponding character from the ciphertext into the correct position in the block:
  - `block[key[j] - 1] = ciphertext.charAt(i + j);`
  - Here, `key[j] - 1` retrieves the position from the key, adjusting for the 0-based indexing of the array. The character from the ciphertext at position `i + j` is assigned to this position in the block.

#### 5. Appending the Rearranged Block:

- After rearranging the characters for the current block, the rearranged block is appended to the `plaintext`.

#### 6. Returning the Result:

- Finally, the method returns the decrypted text as a string.

## ***Dry Run of the code:***

Let's consider a dry run using the following inputs:

- Ciphertext: "Hloel olrWd"

- Key: [3, 1, 2] (This means that the third character goes to the first position, the first character goes to the second position, and the second character goes to the third position.)

#### 1. Initialization:

- plaintext is initialized as an empty string.
- blockSize is set to 3 (the length of the key).

#### 2. Iteration:

- For the first block:

- The loop processes characters from index 0 to 2 (i.e., "Hlo").
- A block array of size 3 is created.
- Rearranging based on the key:

```

▪ For j = 0: block[3 - 1] (i.e., block[2]) = 'H' → block = [' ', '', 'H']
▪ For j = 1: block[1 - 1] (i.e., block[0]) = 'l' → block = ['l', '', 'H']
▪ For j = 2: block[2 - 1] (i.e., block[1]) = 'o' → block = ['l', 'o', 'H']

```

- The rearranged block is "loH" and appended to plaintext, resulting in "loH".

- For the second block:

- The loop processes characters from index 3 to 5 (i.e., "el ").
- A new block array of size 3 is created.
- Rearranging based on the key:

```

▪ For j = 0: block[3 - 1] (i.e., block[2]) = 'e' → block = [' ', '', 'e']
▪ For j = 1: block[1 - 1] (i.e., block[0]) = 'l' → block = ['l', '', 'e']
▪ For j = 2: block[2 - 1] (i.e., block[1]) = ' ' → block = ['l', ' ', 'e']

```

- The rearranged block is "le ", and plaintext becomes "loHle ".

- For the third block:

- The loop processes characters from index 6 to 8 (i.e., "oW").
- The loop attempts to create a block of size 3, but only two characters are available ("oW").
- Rearranging based on the key:

```

▪ For j = 0: block[3 - 1] (i.e., block[2]) = 'o' → block = [' ', '', 'o']
▪ For j = 1: block[1 - 1] (i.e., block[0]) = 'W' → block = ['W', '', 'o']

```

- The rearranged block is "Wo" and appended to plaintext, resulting in "loHle Wo".

#### 3. Final Output:

- The method returns the decrypted text: "loHle Wo".

### Conclusion:

The Permutation Cipher is a straightforward encryption technique that relies on character rearrangement. Decryption involves reversing the rearrangement process using a key that defines the original order of the characters. The provided decryption method effectively restores the original plaintext from the ciphertext by iterating through blocks of characters and rearranging them based on the specified key. While it provides a basic

level of security, it is susceptible to various cryptanalysis techniques and is generally considered insecure by modern standards.

### **Transposition Cipher Decryption:**

#### ***Line-by-Line Explanation of the Code (Implementation):***

The provided code snippet defines a method for decrypting a ciphertext that has been encrypted using the Single Transposition Cipher. It takes two parameters: the encrypted text (ciphertext) and a key (an array of integers that indicates the order of columns). Below is a line-by-line explanation of the code:

1. Initialization:
  - The variable numCols is initialized to the length of the key, which represents the number of columns in the grid.
  - The variable numRows is calculated using the length of the encrypted text divided by numCols, rounded up using Math.ceil. This determines how many rows are needed to accommodate all the characters.
2. Creating the Grid:
  - A 2D character array named grid is initialized with dimensions [numRows][numCols] to hold the characters of the ciphertext.
3. Filling the Grid:
  - An index variable is initialized to zero to track the current position in the encrypted text.
  - A for-each loop iterates over each element in the key:
    - For each key value j, the corresponding column index colIndex is calculated by subtracting 1 (to adjust for 0-based indexing).
    - A nested loop iterates over the rows to fill the current column of the grid:
      - If index is less than the length of the encrypted text, the character at position index in the encrypted text is placed in the grid at the current row and colIndex.
      - The index variable is incremented to move to the next character.
      - If index exceeds the length of the encrypted text, the current grid position is padded with a space to ensure the grid is filled correctly.
4. Building the Decrypted Text:
  - A StringBuilder named decryptedText is initialized to construct the final plaintext.
  - Another nested loop iterates over the rows and columns of the grid:
    - Each character from the grid is appended to decryptedText in row-major order.
5. Returning the Result:
  - Finally, the method returns the decrypted text as a string, using the trim() method to remove any trailing spaces that may have been added during the grid-filling process.

#### ***Dry Run of the code:***

Let's consider a dry run using the following inputs:

- Encrypted Text: "TEHISEXNATG" (The ciphertext)
- Key: [2, 1, 3] (This means that the characters were originally arranged such that the second column comes first, followed by the first column, and then the third column.)



1. Initialization:
  - numCols is set to 3 (the length of the key).
  - numRows is calculated as  $\text{Math.ceil}(10 / 3) = 4$ , meaning 4 rows are needed.
2. Creating the Grid:
  - A grid of size [4][3] is initialized to hold the characters.
3. Filling the Grid:
  - The index is initialized to 0.
  - The loop iterates over the key:
    - For  $j = 2$  (the second column):
      - The characters are placed in grid:
 

- Row 0, Col 1: 'T' → index = 1
        - Row 1, Col 1: 'E' → index = 2
        - Row 2, Col 1: 'H' → index = 3
        - Row 3, Col 1: 'I' → index = 4
    - For  $j = 1$  (the first column):
      - Characters are placed in grid:
 

- Row 0, Col 0: 'S' → index = 5
        - Row 1, Col 0: 'X' → index = 6
        - Row 2, Col 0: 'N' → index = 7
        - Row 3, Col 0: 'A' → index = 8
    - For  $j = 3$  (the third column):
      - Characters are placed in grid:
 

- Row 0, Col 2: 'T' → index = 9
        - Row 1, Col 2: 'G' → index = 10
        - Row 2, Col 2: (pad with space) → index = 10
        - Row 3, Col 2: (pad with space) → index = 10

At this point, the grid looks like this:

```
[ S T T ]
[ X E G ]
[ N H   ]
[ A I   ]
```

4. Building the Decrypted Text:
  - The nested loop reads the grid row-wise:
 

- Row 0: 'S', 'T', 'T' → decryptedText = "STT"
    - Row 1: 'X', 'E', 'G' → decryptedText = "STTXEG"

- Row 2: 'N', 'H', ' ' → decryptedText = "STTXEGNH"
- Row 3: 'A', 'T', ' ' → decryptedText = "STTXEGNHAI"

#### 5. Final Output:

- The method returns the decrypted text after trimming any extra spaces, resulting in "STTXEGNHAI".

### ***Conclusion:***

The Single Transposition Cipher effectively rearranges characters of the plaintext into a grid based on a specified key, which is reversed during decryption. The provided method accomplishes this by filling a grid column-wise and reading it back row-wise, allowing for the original message to be reconstructed. This method highlights a simple yet effective way of encryption and decryption, though it is not considered secure against modern cryptanalysis techniques.

### **Double Transposition Decryption:**

#### ***Line-by-Line Explanation of the Code (Implementation):***

The provided code snippet defines a method for decrypting ciphertext that has been encrypted using the Double Transposition Cipher. It accepts three parameters: the encrypted text (ciphertext), the first key (key1), and the second key (key2). Below is a detailed explanation of the code:

1. Decrypting with the Second Key:
  - The first step involves decrypting the ciphertext using the second key (key2). This is done by calling the `decryptSimpleTranspositionCipher` method, which rearranges the characters in the ciphertext based on the reverse of the second key's order. This produces an intermediate text that is partially decrypted.
2. Decrypting with the First Key:
  - The next step involves taking the intermediate text and decrypting it using the first key (key1). This is again accomplished by calling the `decryptSimpleTranspositionCipher` method, which rearranges the characters in the intermediate text based on the reverse of the first key's order.
3. Returning the Final Decrypted Text:
  - The method returns the final decrypted text, which is the original plaintext that was encrypted using the Double Transposition Cipher.

### ***Dry Run of the code:***

Let's consider a dry run using the following inputs:

- Encrypted Text: "NTHOAEIDPS" (This is the ciphertext).
  - Key 1: [1, 3, 2] (The order for the first transposition).
  - Key 2: [2, 1, 3] (The order for the second transposition).
1. Decrypting with the Second Key:
    - The method begins by decrypting the encrypted text using the second key:
      - For key2 = [2, 1, 3], the `decryptSimpleTranspositionCipher` method is invoked.
      - Following the decryption process as described in the previous example, the intermediate text is rearranged based on the second key's inverse order.
    - Assume after decryption with the second key, the intermediate text results in "HTONAPEIDS".
  2. Decrypting with the First Key:

- The intermediate text "HTONAPEIDS" is then decrypted using the first key:
    - For key1 = [1, 3, 2], the decryptSimpleTranspositionCipher method is called again.
    - The characters are rearranged according to the inverse of the first key's order, resulting in the final decrypted plaintext.
  - Let's assume the result of this step gives "THE PAID SON".
3. Final Output:
- The method returns the final decrypted text "THE PAID SON", which represents the original plaintext.

### ***Conclusion:***

The Double Transposition Cipher enhances security by applying two layers of transposition using two keys. The method for decrypting this cipher effectively reverses the encryption process by first using the second key to produce an intermediate result and then applying the first key to retrieve the original plaintext. This two-step process complicates the decryption for anyone attempting to crack the code without knowledge of both keys.

### **Vigenère Cipher Decryption:**

#### ***Line-by-Line Explanation of the Code (Implementation):***

The provided code snippet defines a method for decrypting ciphertext that has been encrypted using the Vigenère Cipher. It accepts two parameters: the ciphertext to be decrypted and the keyword used for encryption. Here's a detailed breakdown of the code:

1. Initialize Variables:
  - A StringBuilder named plaintext is created to hold the final decrypted text.
  - The keyword is converted to uppercase to maintain consistency in shifting.
2. Iterate Over Each Character in the Ciphertext:
  - The code uses a for loop to iterate through each character in the ciphertext.
3. Character Processing:
  - For each character in the ciphertext (cipherChar), the following checks are made:
    - If cipherChar is a letter:
      - The shift is calculated using the formula:  

$$\text{shift} = (\text{cipherChar} - \text{keyword.charAt}(i \% \text{keyword.length()})) + 26) \% 26$$

$$\text{shift} = (\text{cipherChar} - \text{keyword.charAt}(i \% \text{keyword.length()})) + 26) \% 26$$

$$\text{shift} = (\text{cipherChar} - \text{keyword.charAt}(i \% \text{keyword.length()})) + 26) \% 26$$
      - This formula adjusts the position of the cipherChar by subtracting the corresponding character from the keyword (using the modulo operator to cycle through the keyword).
      - The result is then converted back to a character using:  

$$(\text{char}) (\text{shift} + 'A')$$
    - The decrypted character is appended to the plaintext.
  - If cipherChar is not a letter (e.g., a space, punctuation), it is appended to plaintext unchanged.
4. Return Decrypted Text:
  - After processing all characters, the method returns the complete decrypted text as a string.

*Dry Run of the code:*

Let's consider a dry run using the following inputs:

- Ciphertext: "LXFOPVEFRN" (This is the ciphertext).
  - Keyword: "LEMON".
1. Initial Setup:
    - keyword is converted to uppercase, resulting in "LEMON".
  2. Character Processing:
    - For each character in "LXFOPVEFRN":
      - Index 0:
        - cipherChar = 'L'
        - keyword.charAt(0) = 'L'
        - Shift calculation:  $\text{shift} = (L - L + 26) \bmod 26 = (11 - 11 + 26) \bmod 26 = 0$  \text{shift} =  $(L - L + 26) \bmod 26 = (11 - 11 + 26) \bmod 26 = 0$   
 $\text{shift} = (L - L + 26) \bmod 26 = (11 - 11 + 26) \bmod 26 = 0$
        - Decrypted character = 'A'.
      - Index 1:
        - cipherChar = 'X'
        - keyword.charAt(1) = 'E'
        - Shift calculation:  $\text{shift} = (X - E + 26) \bmod 26 = (23 - 4 + 26) \bmod 26 = 19$  \text{shift} =  $(X - E + 26) \bmod 26 = (23 - 4 + 26) \bmod 26 = 19$   
 $\text{shift} = (X - E + 26) \bmod 26 = (23 - 4 + 26) \bmod 26 = 19$
        - Decrypted character = 'T'.
      - Index 2:
        - cipherChar = 'F'
        - keyword.charAt(2) = 'M'
        - Shift calculation:  $\text{shift} = (F - M + 26) \bmod 26 = (5 - 12 + 26) \bmod 26 = 19$  \text{shift} =  $(F - M + 26) \bmod 26 = (5 - 12 + 26) \bmod 26 = 19$   
 $\text{shift} = (F - M + 26) \bmod 26 = (5 - 12 + 26) \bmod 26 = 19$
        - Decrypted character = 'T'.
      - Index 3:
        - cipherChar = 'O'
        - keyword.charAt(3) = 'O'
        - Shift calculation:  $\text{shift} = (O - O + 26) \bmod 26 = (14 - 14 + 26) \bmod 26 = 0$  \text{shift} =  $(O - O + 26) \bmod 26 = (14 - 14 + 26) \bmod 26 = 0$   
 $\text{shift} = (O - O + 26) \bmod 26 = (14 - 14 + 26) \bmod 26 = 0$
        - Decrypted character = 'A'.
      - Index 4:

- cipherChar = 'P'
  - keyword.charAt(4) = 'N'
  - Shift calculation:  

$$\text{shift} = (P - N + 26) \bmod 26 = (15 - 13 + 26) \bmod 26 = 2$$

$$\text{shift} = (P - N + 26) \bmod 26 = (15 - 13 + 26) \bmod 26 = 2$$
  - Decrypted character = 'C'.
- Index 5:
  - cipherChar = 'V'
  - keyword.charAt(0) = 'L' (wraps around to start of keyword)
  - Shift calculation:  

$$\text{shift} = (V - L + 26) \bmod 26 = (21 - 11 + 26) \bmod 26 = 10$$

$$\text{shift} = (V - L + 26) \bmod 26 = (21 - 11 + 26) \bmod 26 = 10$$
  - Decrypted character = 'K'.
- Index 6:
  - cipherChar = 'E'
  - keyword.charAt(1) = 'E'
  - Shift calculation:  $\text{shift} = (E - E + 26) \bmod 26 = (4 - 4 + 26) \bmod 26 = 0$   

$$\text{shift} = (E - E + 26) \bmod 26 = (4 - 4 + 26) \bmod 26 = 0$$
  - Decrypted character = 'A'.
- Index 7:
  - cipherChar = 'F'
  - keyword.charAt(2) = 'M'
  - Shift calculation:  

$$\text{shift} = (F - M + 26) \bmod 26 = (5 - 12 + 26) \bmod 26 = 19$$

$$\text{shift} = (F - M + 26) \bmod 26 = (5 - 12 + 26) \bmod 26 = 19$$
  - Decrypted character = 'T'.
- Index 8:
  - cipherChar = 'R'
  - keyword.charAt(3) = 'O'
  - Shift calculation:  

$$\text{shift} = (R - O + 26) \bmod 26 = (17 - 14 + 26) \bmod 26 = 3$$

$$\text{shift} = (R - O + 26) \bmod 26 = (17 - 14 + 26) \bmod 26 = 3$$
  - Decrypted character = 'D'.
- Index 9:
  - cipherChar = 'N'
  - keyword.charAt(4) = 'N'

- Shift calculation:  

$$\text{shift} = (N - N + 26) \bmod 26 = (13 - 13 + 26) \bmod 26 = 0$$

$$\text{shift} = (N - N + 26) \bmod 26 = (13 - 13 + 26) \bmod 26 = 0$$

$$\text{shift} = (N - N + 26) \bmod 26 = (13 - 13 + 26) \bmod 26 = 0$$
- Decrypted character = 'A'.

### 3. Final Output:

- The final decrypted text after processing all characters is "ATTACKATDAWN".

### ***Conclusion:***

The Vigenère Cipher is a powerful method of encryption that enhances security through the use of a keyword. The decryption method effectively reverses the encryption process by applying shifts based on the keyword to recover the original plaintext. By handling non-letter characters appropriately, the method ensures that the integrity of the message is maintained during the decryption process.

### **Decryption using DES and 3DES Algorithm:**

#### ***Line-by-Line Explanation of the Code (Implementation):***

The provided code implements a method to decrypt ciphertext that was encrypted using the DES algorithm in ECB (Electronic Codebook) mode. It takes two parameters: the Base64-encoded ciphertext and the secret key used for decryption.

1. **Importing Required Classes:** The necessary imports include classes such as `javax.crypto.Cipher`, `javax.crypto.SecretKey`, and `java.util.Base64`.
2. **Method Declaration:** The method is declared as public static, which means it can be accessed without creating an instance of the class. It returns the decrypted text in the form of a string and throws exceptions if any decryption-related errors occur.
3. **Create a Cipher Object:** The Cipher object is created and configured to use the DES algorithm in ECB mode with PKCS5 padding. The "DES/ECB/PKCS5Padding" string defines the encryption algorithm, mode, and padding scheme.
4. **Initialize the Cipher for Decryption:** The cipher is initialized in decryption mode, using the secret key provided. This sets up the cipher to process the ciphertext for decryption.
5. **Decrypt the Ciphertext:** The Base64-encoded ciphertext is first decoded into a byte array. The `doFinal` method is then used to decrypt the byte array into plaintext bytes.
6. **Convert Decrypted Bytes to String:** Finally, the decrypted byte array is converted back into a string, which is returned as the method's output, representing the original plaintext before encryption.

### 3DES Decryption Explanation:

#### ***Line-by-Line Explanation of the Code (Implementation):***

This code implements a method for decrypting ciphertext that was encrypted using the 3DES (Triple DES) algorithm in ECB mode. It takes two parameters: the Base64-encoded ciphertext and the secret key used for decryption.

1. **Importing Required Classes:** The necessary imports include classes such as `javax.crypto.Cipher`, `javax.crypto.SecretKey`, and `java.util.Base64`.
2. **Method Declaration:** The method is declared as public static, meaning it can be accessed without an instance of the class. It returns the decrypted plaintext as a string and throws exceptions if any decryption errors occur.
3. **Create a Cipher Object:** A Cipher object is created and configured to use the 3DES algorithm (also called DESede) in ECB mode with PKCS5 padding. The "DESede/ECB/PKCS5Padding" string specifies the encryption algorithm, mode, and padding scheme.

4. Initialize the Cipher for Decryption: The cipher is initialized in decryption mode, using the provided secret key. This prepares the cipher to process the encrypted data.
5. Decrypt the Ciphertext: The ciphertext, which is Base64-encoded, is decoded into a byte array. The doFinal method of the cipher is called to decrypt the byte array into plaintext bytes.
6. Convert Decrypted Bytes to String: The decrypted byte array is then converted into a string and returned, representing the original message before it was encrypted using 3DES.

### ***Conclusion:***

Both the DES and 3DES decryption methods follow similar processes. They take Base64-encoded ciphertext and decrypt it using a secret key. While DES uses a single key for encryption and decryption, 3DES (DESede) applies the DES algorithm three times with different keys, making it more secure than DES. Both methods decrypt the input into byte arrays and convert the result into a readable string.

### **AES Decryption Using ECB:**

#### ***Line-by-Line Explanation of the Code (Implementation):***

The provided code snippet defines a method for decrypting ciphertext that has been encrypted using AES in ECB mode. It takes two parameters: the Base64-encoded ciphertext to be decrypted and the secret key used for decryption. Here's a line-by-line explanation of the code:

1. Importing Required Classes:
  - Although not shown in the snippet, necessary imports include javax.crypto.Cipher, javax.crypto.SecretKey, and java.util.Base64.
2. Method Declaration:
  - The method is declared as public static String decryptAESUsingECBMode(String ciphertext, SecretKey key) throws Exception. It indicates that this method can throw exceptions.
3. Create a Cipher Object:
  - Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
  - This line creates a Cipher object configured to use the AES algorithm in ECB mode with PKCS5 padding. Padding is necessary when the length of the plaintext is not a multiple of the block size (16 bytes for AES).
4. Initialize the Cipher for Decryption:
  - cipher.init(Cipher.DECRYPT\_MODE, key);
  - This line initializes the cipher object for decryption using the provided SecretKey. The cipher will now be set to decrypt mode.
5. Decrypt the Ciphertext:
  - byte[] decryptedBytes = cipher.doFinal(Base64.getDecoder().decode(ciphertext));
  - The method first decodes the Base64-encoded ciphertext into bytes.
  - Then, it calls doFinal() to perform the decryption, which processes the decoded bytes and returns the decrypted data as a byte array.
6. Convert Decrypted Bytes to String:
  - return new String(decryptedBytes);
  - Finally, this line converts the decrypted byte array back into a string, which is returned as the output of the method.

### ***Dry Run of the code:***

Let's consider a dry run using hypothetical inputs:

- Ciphertext: "U2FsdGVkX19kWxTz9rE1tnq+NBhV5gUWa1eZyU6hReo=" (Base64-encoded ciphertext)
  - Key: An example `SecretKey` object (we'll assume it has been generated and passed to the method).
1. Creating the Cipher Object:
    - The method creates a cipher object configured for AES in ECB mode with PKCS5 padding.
  2. Initializing the Cipher for Decryption:
    - The cipher is initialized with the secret key.
  3. Decrypting the Ciphertext:
    - The Base64-encoded ciphertext is decoded:
      - Decoded bytes: `byte[] decodedBytes = Base64.getDecoder().decode("U2FsdGVkX19kWxTz9rE1tnq+NBhV5gUWa1eZyU6hReo=");`
    - The `doFinal()` method is called with the decoded bytes:
      - `byte[] decryptedBytes = cipher.doFinal(decodedBytes);`
    - Assume the resulting decrypted bytes are `decryptedBytes = "Hello, World!".getBytes();` (this is a hypothetical result).
  4. Converting Decrypted Bytes to String:
    - The byte array is converted back to a string:
      - `return new String(decryptedBytes);`
    - Final output: "Hello, World!".

### ***Conclusion:***

The method effectively implements AES decryption in ECB mode, handling Base64-encoded ciphertext and using a secret key for the decryption process. While it demonstrates the essential steps involved in AES decryption, it's important to note that due to the weaknesses of ECB mode, alternatives such as CBC (Cipher Block Chaining) mode are generally recommended for practical applications to enhance security and avoid revealing patterns in the data.

### **AES Decryption Using CBC:**

#### ***Line-by-Line Explanation of the Code (Implementation):***

The provided code snippet implements a method for decrypting ciphertext encrypted using AES in CBC mode. The method takes three parameters: the Base64-encoded ciphertext, the secret key for decryption, and the IV used during encryption. Here's a detailed explanation of the code:

1. Importing Required Classes:
  - While not included in the snippet, necessary imports typically include `javax.crypto.Cipher`, `javax.crypto.SecretKey`, `javax.crypto.spec.IvParameterSpec`, and `java.util.Base64`.
2. Method Declaration:
  - The method is defined as `public static String decryptAESUsingCBC(String ciphertext, SecretKey key, byte[] iv) throws Exception`, indicating it can throw exceptions.
3. Create a Cipher Object:
  - `Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");`



- This line creates a Cipher object configured to use the AES algorithm in CBC mode with PKCS5 padding. The padding is necessary to handle cases where the plaintext's length is not a multiple of the block size (16 bytes for AES).
4. Create IV Parameter Spec:
    - `IvParameterSpec ivParams = new IvParameterSpec(iv);`
    - This line initializes an `IvParameterSpec` object using the provided IV. This specification is crucial for the decryption process, as it ensures that the cipher operates correctly with the same IV used during encryption.
  5. Initialize the Cipher for Decryption:
    - `cipher.init(Cipher.DECRYPT_MODE, key, ivParams);`
    - This line initializes the cipher for decryption using the provided `SecretKey` and `IvParameterSpec`. The cipher is now set to decrypt mode with the specified IV.
  6. Decrypt the Ciphertext:
    - `byte[] decryptedBytes = cipher.doFinal(Base64.getDecoder().decode(ciphertext));`
    - The method first decodes the Base64-encoded ciphertext into bytes.
    - It then calls `doFinal()` to perform the decryption, processing the decoded bytes and returning the decrypted data as a byte array.
  7. Convert Decrypted Bytes to String:
    - `return new String(decryptedBytes);`
    - Finally, this line converts the decrypted byte array back into a string, which is returned as the method's output.

### *Dry Run of the code:*

Let's consider a hypothetical example using the following inputs:

- Ciphertext: "DgLcnjU8Y6c6uXYrM54MDA==" (Base64-encoded ciphertext)
  - Key: An example `SecretKey` object (assumed to be generated and passed to the method).
  - IV: A byte array initialized with the same IV used during encryption, e.g., `byte[] iv = new byte[16];`.
1. Creating the Cipher Object:
    - The method creates a cipher object configured for AES in CBC mode with PKCS5 padding.
  2. Creating IV Parameter Spec:
    - An `IvParameterSpec` object is initialized with the IV.
  3. Initializing the Cipher for Decryption:
    - The cipher is initialized with the secret key and the IV.
  4. Decrypting the Ciphertext:
    - The Base64-encoded ciphertext is decoded:
      - `byte[] decodedBytes = Base64.getDecoder().decode("DgLcnjU8Y6c6uXYrM54MDA==");`
    - The `doFinal()` method is called with the decoded bytes:
      - `byte[] decryptedBytes = cipher.doFinal(decodedBytes);`

- Assume the resulting decrypted bytes are `decryptedBytes = "Secret Message!".getBytes();` (this is a hypothetical result).
5. Converting Decrypted Bytes to String:
- The byte array is converted back to a string:
    - `return new String(decryptedBytes);`
  - Final output: "Secret Message!".

### ***Conclusion:***

The method successfully implements AES decryption in CBC mode, utilizing Base64-encoded ciphertext and an IV alongside a secret key for the decryption process. This code illustrates how CBC mode adds an important layer of security compared to simpler modes like ECB by incorporating an IV, thus preventing pattern recognition in the ciphertext. It is crucial to manage the IV carefully, ensuring that it is unique and not reused with the same key to maintain the encryption's integrity.

### **AES Decryption Using CFB:**

#### ***Line-by-Line Explanation of the Code (Implementation):***

The code provided implements a method for decrypting ciphertext that has been encrypted using AES in CFB mode. It accepts three parameters: the Base64-encoded ciphertext, the secret key for decryption, and the Initialization Vector (IV) used during encryption. Here's a detailed line-by-line explanation of the code:

1. Importing Required Classes:
  - Necessary imports include classes such as `javax.crypto.Cipher`, `javax.crypto.SecretKey`, `javax.crypto.spec.IvParameterSpec`, and `java.util.Base64`.
2. Method Declaration:
  - The method is defined as `public static String decryptAESUsingCFB(String ciphertext, SecretKey key, byte[] iv) throws Exception`. The `throws Exception` indicates that this method may throw exceptions, such as those related to encryption or decryption.
3. Create a Cipher Object:
  - `Cipher cipher = Cipher.getInstance("AES/CFB/PKCS5Padding");`
  - This line creates a `Cipher` object that is set up to use the AES algorithm in CFB mode with PKCS5 padding. Padding is not strictly necessary for CFB mode, but it can be included to maintain compatibility.
4. Create IV Parameter Spec:
  - `IvParameterSpec ivParams = new IvParameterSpec(iv);`
  - This line initializes an `IvParameterSpec` object with the provided IV. The IV is essential for the CFB decryption process as it ensures that the decryption corresponds correctly with the encryption that used the same IV.
5. Initialize the Cipher for Decryption:
  - `cipher.init(Cipher.DECRYPT_MODE, key, ivParams);`
  - Here, the cipher is initialized for decryption. The `SecretKey` and `IvParameterSpec` are passed to the `init` method, preparing the cipher for the decryption operation.
6. Decrypt the Ciphertext:
  - `byte[] decryptedBytes = cipher.doFinal(Base64.getDecoder().decode(ciphertext));`
  - The Base64-encoded ciphertext is decoded into a byte array.

- The doFinal() method performs the decryption on the decoded bytes, returning the decrypted data as a byte array.
7. Convert Decrypted Bytes to String:
    - return new String(decryptedBytes);
    - Finally, this line converts the decrypted byte array back into a string, which is returned as the method's output.

### ***Dry Run of the code:***

Let's consider a hypothetical example using the following inputs:

- Ciphertext: "E5ZyI8LkGmM2dm6M5hT1QQ==" (Base64-encoded ciphertext).
- Key: A SecretKey object (assumed to be generated and passed to the method).
- IV: A byte array initialized with the same IV used during encryption, e.g., byte[] iv = new byte[16];.

1. Creating the Cipher Object:
  - The method creates a cipher object configured for AES in CFB mode with PKCS5 padding.
2. Creating IV Parameter Spec:
  - An IvParameterSpec object is initialized with the IV.
3. Initializing the Cipher for Decryption:
  - The cipher is initialized with the secret key and the IV.
4. Decrypting the Ciphertext:
  - The Base64-encoded ciphertext is decoded:
    - byte[] decodedBytes = Base64.getDecoder().decode("E5ZyI8LkGmM2dm6M5hT1QQ==");
  - The doFinal() method is called with the decoded bytes:
    - byte[] decryptedBytes = cipher.doFinal(decodedBytes);
  - Assume the resulting decrypted bytes are decryptedBytes = "Secret Message!".getBytes(); (this is a hypothetical result).
5. Converting Decrypted Bytes to String:
  - The byte array is converted back to a string:
    - return new String(decryptedBytes);
  - Final output: "Secret Message!".

### ***Conclusion:***

The provided method implements the decryption of ciphertext encrypted with AES in CFB mode. This approach allows the processing of arbitrary lengths of data, making it versatile for various applications. Using an IV enhances security by ensuring that identical plaintext blocks produce different ciphertexts, thereby avoiding predictable patterns in the encrypted output. CFB mode, with its ability to handle streaming data, is useful in scenarios where data is processed in smaller units rather than in fixed block sizes, providing flexibility in encryption and decryption operations.

### **AES Decryption Using OFB:**

#### ***Line-by-Line Explanation of the Code (Implementation):***

The provided code implements a method to decrypt ciphertext that was encrypted using AES in OFB (Output Feedback) mode. It takes three parameters: the Base64-encoded ciphertext, the secret key for decryption, and the Initialization Vector (IV) used during encryption. Below is a detailed explanation of each line:

1. **Importing Required Classes:** The necessary imports include classes such as `javax.crypto.Cipher`, `javax.crypto.SecretKey`, `javax.crypto.spec.IvParameterSpec`, and `java.util.Base64`.
2. **Method Declaration:** The method is declared as a public static method, which means it can be called without an instance of the class. It returns a string after decryption and throws an exception if there are any encryption or decryption-related errors.
3. **Create a Cipher Object:** A Cipher object is created and configured to use the AES algorithm in OFB mode with PKCS5 padding. The "AES/OFB/PKCS5Padding" string defines the mode and padding used in the decryption process.
4. **Create IV Parameter Spec:** An `IvParameterSpec` object is initialized using the provided IV (Initialization Vector). This IV is necessary for decryption, as OFB mode requires the same IV that was used during encryption to produce the correct decrypted output.
5. **Initialize the Cipher for Decryption:** The cipher is initialized in decryption mode using the secret key and IV. This prepares the cipher to process the incoming ciphertext for decryption.
6. **Decrypt the Ciphertext:** The Base64-encoded ciphertext is decoded into a byte array, and the `doFinal` method of the cipher is used to perform the actual decryption. The result is a byte array containing the decrypted data.
7. **Convert Decrypted Bytes to String:** Finally, the decrypted byte array is converted back into a string and returned as the method's output. This string represents the original plaintext before it was encrypted.

### *Dry Run of the Code:*

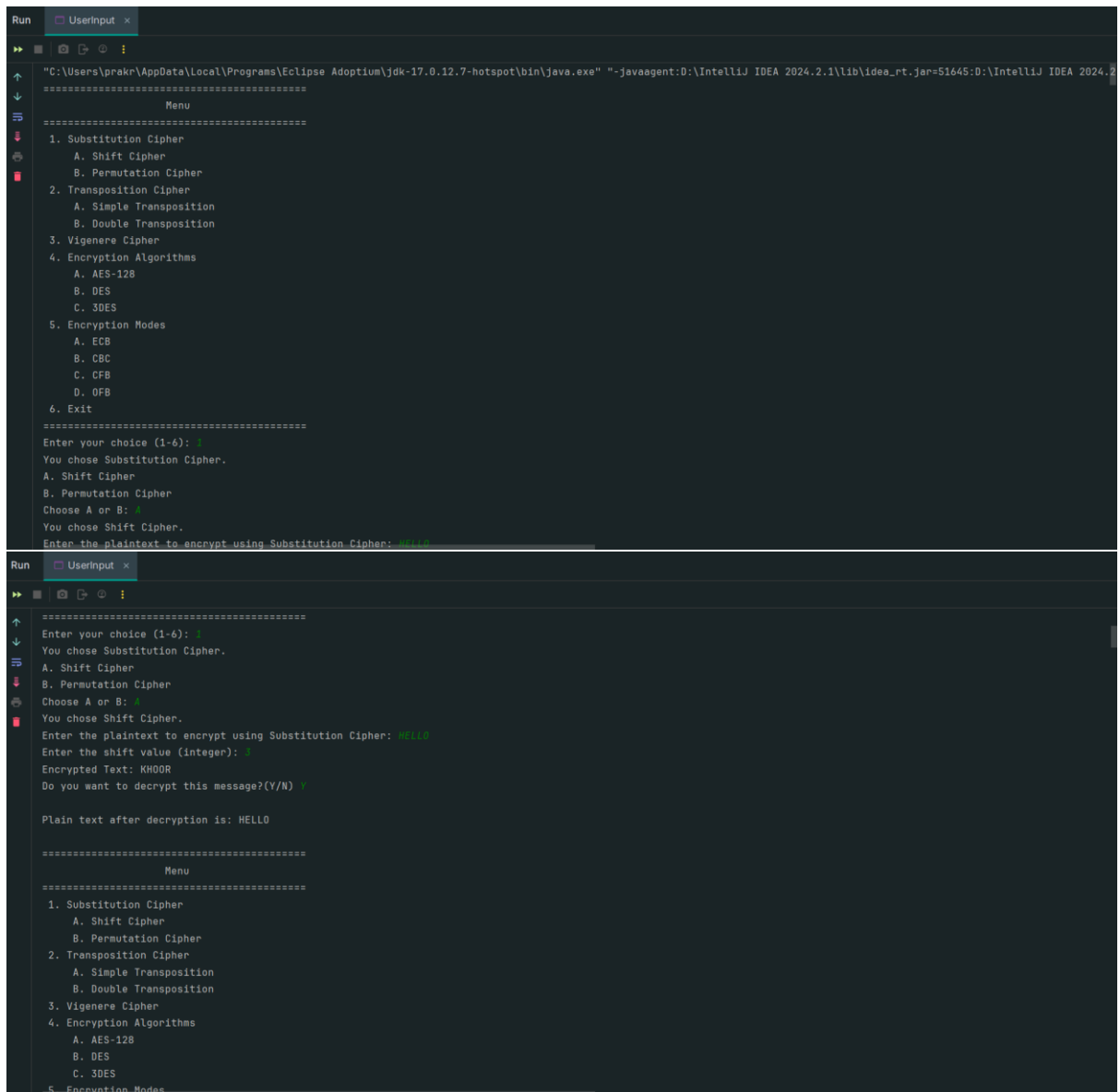
Let's consider an example using the following inputs:

- Ciphertext: "E5ZyI8LkGmM2dm6M5hT1QQ==" (a hypothetical Base64-encoded ciphertext).
  - Key: A secret key object (assumed to have been generated and passed to the method).
  - IV: A byte array initialized with the same IV used during encryption.
1. **Creating the Cipher Object:** The method creates a cipher object configured for AES in OFB mode with PKCS5 padding.
  2. **Creating IV Parameter Spec:** An `IvParameterSpec` object is initialized with the provided IV.
  3. **Initializing the Cipher for Decryption:** The cipher is initialized with the secret key and the IV to prepare for decryption.
  4. **Decrypting the Ciphertext:** The Base64-encoded ciphertext is decoded into a byte array. The `doFinal` method is then called to decrypt the byte array into plaintext bytes.
  5. **Converting Decrypted Bytes to String:** The byte array containing the decrypted data is converted back into a string, which represents the original plaintext.

### *Conclusion:*

This method decrypts ciphertext that has been encrypted using AES in OFB mode. OFB mode is a feedback-based encryption mode that can handle any data size, making it useful for streaming data. The use of an IV enhances security by ensuring that identical plaintexts produce different ciphertexts, preventing predictable patterns in the output. The method effectively decrypts the data using the same secret key and IV used during encryption, and it returns the original plaintext.

**Screenshots of working code and all Ciphers/Algorithms/Modes for Encryption and Decryption(both):**



```
Run  UserInput x
C:\Users\prakr\AppData\Local\Programs\Eclipse Adoptium\jdk-17.0.12-hotspot\bin\java.exe" "-javaagent:D:\IntelliJ IDEA 2024.2.1\lib\idea_rt.jar=51645:D:\IntelliJ IDEA 2024.2
=====
Menu
=====
1. Substitution Cipher
   A. Shift Cipher
   B. Permutation Cipher
2. Transposition Cipher
   A. Simple Transposition
   B. Double Transposition
3. Vigenere Cipher
4. Encryption Algorithms
   A. AES-128
   B. DES
   C. 3DES
5. Encryption Modes
   A. ECB
   B. CBC
   C. CFB
   D. OFB
6. Exit
=====
Enter your choice (1-6): 1
You chose Substitution Cipher.
A. Shift Cipher
B. Permutation Cipher
Choose A or B: A
You chose Shift Cipher.
Enter the plaintext to encrypt using Substitution Cipher: HELLO

Run  UserInput x
=====
Enter your choice (1-6): 1
You chose Substitution Cipher.
A. Shift Cipher
B. Permutation Cipher
Choose A or B: A
You chose Shift Cipher.
Enter the plaintext to encrypt using Substitution Cipher: HELLO
Enter the shift value (integer): 3
Encrypted Text: KHOOR
Do you want to decrypt this message?(Y/N) Y

Plain text after decryption is: HELLO

=====
Menu
=====
1. Substitution Cipher
   A. Shift Cipher
   B. Permutation Cipher
2. Transposition Cipher
   A. Simple Transposition
   B. Double Transposition
3. Vigenere Cipher
4. Encryption Algorithms
   A. AES-128
   B. DES
   C. 3DES
5. Encryption Modes
```



```

Run  UserInput x
B. Double Transposition
3. Vigenere Cipher
4. Encryption Algorithms
  A. AES-128
  B. DES
  C. 3DES
5. Encryption Modes
  A. ECB
  B. CBC
  C. CFB
  D. OFB
6. Exit
=====
Enter your choice (1-6): 2
You chose Transposition Cipher.
  A. Simple Transposition
  B. Double Transposition
Choose A or B: 2
You chose Double Transposition.
Enter the plaintext to encrypt using Double Transposition Cipher: HELLO WORLD
Enter the first key (as comma-separated integers): 1,1,4,2
Enter the second key (as comma-separated integers): 2,1,3,4
Encrypted text: WRELO DL HOL
Do you want to decrypt this message?(Y/N) Y

Plain text after decryption is: HELLO WORLD

=====
Menu
ignment_IJ > src > UserInput > main 12:24 CRLF UTF-8 4 spaces

Run  UserInput x
B. Permutation Cipher
2. Transposition Cipher
  A. Simple Transposition
  B. Double Transposition
3. Vigenere Cipher
4. Encryption Algorithms
  A. AES-128
  B. DES
  C. 3DES
5. Encryption Modes
  A. ECB
  B. CBC
  C. CFB
  D. OFB
6. Exit
=====
Enter your choice (1-6): 3
You chose Vigenere Cipher.
Enter the plaintext to encrypt using Vigenere Cipher: HELLO WORLD
Enter the keyword: KEY
Encrypted Text: RIJVS GSPVH
Do you want to decrypt this message?(Y/N) Y

Plain text after decryption is: HELLO WORLD

=====
Menu
1. Substitution Cipher
ignment_IJ > src > UserInput > main 12:24 CRLF UTF-8 4 spaces

Run  UserInput x
B. Double Transposition
3. Vigenere Cipher
4. Encryption Algorithms
  A. AES-128
  B. DES
  C. 3DES
5. Encryption Modes
  A. ECB
  B. CBC
  C. CFB
  D. OFB
6. Exit
=====
Enter your choice (1-6): 4
You chose Encryption Algorithms.
  A. AES-128
  B. DES
  C. 3DES
Choose A, B, or C: A
Enter the text you want to encrypt using the chosen algorithm: HELLOIAMPRAKRITIISHARMA
Do you want to enter a custom key? (Y/N): Y
Enter your 16-character key for AES: 12345678901234567
Encrypted Text (AES-128): eMps6z2mBCnHRNDNBb0BbhMqxtYSd6Q9l0EylUaPPCQ=
Do you want to decrypt this message? (Y/N): Y
Plain text after decryption is: HELLOIAMPRAKRITIISHARMA

=====
Menu
=====

```

```

Run  UserInput x
>>  A. Shift Cipher
    B. Permutation Cipher
    2. Transposition Cipher
      A. Simple Transposition
      B. Double Transposition
    3. Vigenere Cipher
    4. Encryption Algorithms
      A. AES-128
      B. DES
      C. 3DES
    5. Encryption Modes
      A. ECB
      B. CBC
      C. CFB
      D. OFB
    6. Exit
=====
Enter your choice (1-6): 4
You chose Encryption Algorithms.
    A. AES-128
    B. DES
    C. 3DES
Choose A, B, or C: 4
Enter the text you want to encrypt using the chosen algorithm: HELLOIAMPRAKRITISHARMA
Do you want to enter a custom key? (Y/N): N
Encrypted Text (AES-128): pycOp2ML/1w2z4Kd7DoAAv0cMIH5Qq2PUwu0Apzax8=
Do you want to decrypt this message? (Y/N): Y
Plain text after decryption is: HELLOIAMPRAKRITISHARMA

Run  UserInput x
>>  A. Simple Transposition
    B. Double Transposition
    3. Vigenere Cipher
    4. Encryption Algorithms
      A. AES-128
      B. DES
      C. 3DES
    5. Encryption Modes
      A. ECB
      B. CBC
      C. CFB
      D. OFB
    6. Exit
=====
Enter your choice (1-6): 4
You chose Encryption Algorithms.
    A. AES-128
    B. DES
    C. 3DES
Choose A, B, or C: 4
Enter the text you want to encrypt using the chosen algorithm: HELLOIAMPRAKRITISHARMA
Do you want to enter a custom key? (Y/N): Y
Enter your 8-character key for DES: 12345678
Encrypted Text (DES): uG5qEz0cMYUQNSpQdmFWre0Rsyb+g2Ia
Do you want to decrypt this message? (Y/N): Y
Plain text after decryption is: HELLOIAMPRAKRITISHARMA

=====
Menu

Assignment_II > src > UserInput > main 12:24 CRLF UTF-8 4 spaces

Run  UserInput x
>>  B. DES
    C. 3DES
    5. Encryption Modes
      A. ECB
      B. CBC
      C. CFB
      D. OFB
    6. Exit
=====
Enter your choice (1-6): 4
You chose Encryption Algorithms.
    A. AES-128
    B. DES
    C. 3DES
Choose A, B, or C: B
Enter the text you want to encrypt using the chosen algorithm: HELLOIAMPRAKRITISHARMA
Do you want to enter a custom key? (Y/N): N
Encrypted Text (DES): YW+P+ytles4ASoSJUJna5fmNxxQvHHVw
Do you want to decrypt this message? (Y/N): Y
Plain text after decryption is: HELLOIAMPRAKRITISHARMA

=====
Menu

=====
1. Substitution Cipher
  A. Shift Cipher
  B. Permutation Cipher
2. Transposition Cipher
  A. Simple Transposition

```



```

Run  UserInput x
>> B. DES
C. 3DES
5. Encryption Modes
A. ECB
B. CBC
C. CFB
D. OFB
6. Exit
=====
Enter your choice (1-6): 4
You chose Encryption Algorithms.
A. AES-128
B. DES
C. 3DES
Choose A, B, or C: C
Enter the text you want to encrypt using the chosen algorithm: HELLOIAMPRAKRITISHARMAFROMINDIA
Do you want to enter a custom key? (Y/N): Y
Enter your 8-character key for DESede: 123456789123456
Key must be exactly 8 characters long.

=====
Menu
=====
1. Substitution Cipher
A. Shift Cipher
B. Permutation Cipher
2. Transposition Cipher
A. Simple Transposition
B. Double Transposition

Run  UserInput x
>> B. Double Transposition
3. Vigenere Cipher
4. Encryption Algorithms
A. AES-128
B. DES
C. 3DES
5. Encryption Modes
A. ECB
B. CBC
C. CFB
D. OFB
6. Exit
=====
Enter your choice (1-6): 4
You chose Encryption Algorithms.
A. AES-128
B. DES
C. 3DES
Choose A, B, or C: C
Enter the text you want to encrypt using the chosen algorithm: HELLOIAMPRAKRITISHARMA
Do you want to enter a custom key? (Y/N): Y
Encrypted Text (3DES): e0s7bHS4SHLXxv0VhAI6IhLx7UuU0QD0
Do you want to decrypt this message? (Y/N): Y
Plain text after decryption is: HELLOIAMPRAKRITISHARMA

=====
Menu
=====
1. Substitution Cipher

signment_JI > src > UserInput > main 12:24 CRLF UTF-8 4 spaces

Project  Functionalities.java  UserInput.java x
Run  UserInput x
>> A. AES-128
B. DES
C. 3DES
5. Encryption Modes
A. ECB
B. CBC
C. CFB
D. OFB
6. Exit
=====
Enter your choice (1-6): 5
You chose Encryption Modes.
A. ECB
B. CBC
C. CFB
D. OFB
Choose A, B, C or D: A
Enter the plaintext to encrypt: HELLOIAMPRAKRITISHARMA
Do you want to enter a custom key? (Y/N): N
Encrypted Text: pyc0pz2ML/1w2z4Kd7DoAAv0cMIH5Qq2PUwu0Apzax0=
Do you want to decrypt this message?(Y/N) Y

Plain text after decryption is: HELLOIAMPRAKRITISHARMA
Encrypted Text: pyc0pz2ML/1w2z4Kd7DoAAv0cMIH5Qq2PUwu0Apzax0=

=====
Menu
=====
1. Substitution Cipher

signment_JI > src > UserInput > main 12:24 CRLF UTF-8 4 spaces

```

```

Run  UserInput x
>> 5. Encryption Modes
A. ECB
B. CBC
C. CFB
D. OFB
6. Exit
=====
Enter your choice (1-6): 5
You chose Encryption Modes.
A. ECB
B. CBC
C. CFB
D. OFB
Choose A, B, C or D: 5
Enter the plaintext to encrypt: HELLOIAMPRAKRITISHARMA
Do you want to enter a custom key? (Y/N): N
Encrypted Text: uL37a/21/loy+WU2suLFkeSjmltM9m6/VdTL1GWowA=
Do you want to decrypt this message?(Y/N) Y

Plain text after decryption is: HELLOIAMPRAKRITISHARMA
Encrypted Text: uL37a/21/loy+WU2suLFkeSjmltM9m6/VdTL1GWowA=

=====
Menu
=====
1. Substitution Cipher
A. Shift Cipher
B. Permutation Cipher
2. Transposition Cipher

Assignment_II > src > UserInput > main 12:24 CRLF UTF-8 4 spaces

Run  UserInput x
>> A. AES-128
B. DES
C. 3DES
5. Encryption Modes
A. ECB
B. CBC
C. CFB
D. OFB
6. Exit
=====
Enter your choice (1-6): 5
You chose Encryption Modes.
A. ECB
B. CBC
C. CFB
D. OFB
Choose A, B, C or D: C
Enter the plaintext to encrypt: HELLOIAMPRAKRITISHARMA
Do you want to enter a custom key? (Y/N): Y
Encrypted Text: 4lTndBjdl+j4bFTPexHZZ+45BPwdKT046jXQl3t0NzH=
Do you want to decrypt this message?(Y/N) Y

Plain text after decryption is: HELLOIAMPRAKRITISHARMA
Encrypted Text: 4lTndBjdl+j4bFTPexHZZ+45BPwdKT046jXQl3t0NzH=

=====
Menu
=====
1. Substitution Cipher

Assignment_II > src > UserInput > main 12:24 CRLF UTF-8 4 spaces

Run  UserInput x
>> B. Double Transposition
3. Vigenere Cipher
4. Encryption Algorithms
A. AES-128
B. DES
C. 3DES
5. Encryption Modes
A. ECB
B. CBC
C. CFB
D. OFB
6. Exit
=====
Enter your choice (1-6): 5
You chose Encryption Modes.
A. ECB
B. CBC
C. CFB
D. OFB
Choose A, B, C or D: Enter the plaintext to encrypt: 0
Do you want to enter a custom key? (Y/N): Y
Invalid choice. Please choose A, B, C or D.

=====
Menu
=====
1. Substitution Cipher
A. Shift Cipher

```

```

Run  UserInput x
>> [Icons]
↑
↓
5. Encryption Modes
  A. ECB
  B. CBC
  C. CFB
  D. OFB
6. Exit
=====
Enter your choice (1-6): 5
You chose Encryption Modes.
  A. ECB
  B. CBC
  C. CFB
  D. OFB
Choose A, B, C or D: 5
Enter the plaintext to encrypt: HELLOIAMPRAKRITISHARMA
Do you want to enter a custom key? (Y/N): 5
Encrypted Text: S0L1he8bZjcBpJHjfHY/eHxN096J9Vc2K2fGqvUqVzk=
Do you want to decrypt this message?(Y/N) 5

Plain text after decryption is: HELLOIAMPRAKRITISHARMA
Encrypted Text: S0L1he8bZjcBpJHjfHY/eHxN096J9Vc2K2fGqvUqVzk=

=====
Menu
=====
1. Substitution Cipher
  A. Shift Cipher

signment_II > src > UserInput > main 12:24 CRLF UTF-8 4 spaces

Run  UserInput x
>> [Icons]
↑
↓
=====
Menu
=====
1. Substitution Cipher
  A. Shift Cipher
  B. Permutation Cipher
2. Transposition Cipher
  A. Simple Transposition
  B. Double Transposition
3. Vigenere Cipher
4. Encryption Algorithms
  A. AES-128
  B. DES
  C. 3DES
5. Encryption Modes
  A. ECB
  B. CBC
  C. CFB
  D. OFB
6. Exit
=====
Enter your choice (1-6): 5
Exiting...

Process finished with exit code 0

signment_II > src > UserInput > main 12:24 CRLF UTF-8 4 spaces

```

**CODE****Functionalities.java**

```

import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;
import java.security.SecureRandom;
import java.util.Base64;
import java.util.Scanner;

public class Functionalities {

    // 8 characters for 3DES (used the same as DES)

    // ***** MENU HANDLING METHODS WITH SWITCH CASES *****

    // Handling the first substitution Cipher choice
    static void handleSubstitutionCipher(Scanner scanner) {
        System.out.println("You chose Substitution Cipher.");
        System.out.println("A. Shift Cipher");
        System.out.println("B. Permutation Cipher");
        System.out.print("Choose A or B: ");
        String subChoice = scanner.nextLine().toUpperCase();
        String decryptionChoice;
        switch (subChoice) {
            case "A":
                System.out.println("You chose Shift Cipher.");
                System.out.print("Enter the plaintext to encrypt using Substitution Cipher: ");
                String plaintextShift = scanner.nextLine();
                System.out.print("Enter the shift value (integer): ");
                int shiftValue = scanner.nextInt();
                scanner.nextLine(); // Consume newline character
                String encryptedShift = shiftEncryptMethod(plaintextShift, shiftValue);
                System.out.println("Encrypted Text: " + encryptedShift);
                System.out.print("Do you want to decrypt this message?(Y/N) ");
                decryptionChoice = scanner.nextLine();
                scanner.nextLine(); // Consume newline character
                switch (decryptionChoice) {
                    case "Y":
                        System.out.println("Plain text after decryption is: " + decryptShiftCipher(encryptedShift, shiftValue));
                        break;
                    case "N":
                        break;
                }
            break;

            case "B":
                System.out.println("You chose Permutation Cipher.");
                System.out.print("Enter the plaintext to encrypt using Permutation Cipher: ");
                String plaintextPermutation = scanner.nextLine();
                System.out.print("Enter the permutation key (as comma-separated integers): ");
                String keyInput = scanner.nextLine();
                int[] permutationKey = parseKeyInput(keyInput);
                String encryptedPermutation = permutationEncryptMethod(plaintextPermutation, permutationKey);

```

```

        System.out.println("Encrypted text: " + encryptedPermutation);
        System.out.print("Do you want to decrypt this message?(Y/N) ");
        decryptionChoice = scanner.nextLine();
        scanner.nextLine(); // Consume newline character
        switch (decryptionChoice) {
            case "Y":
                System.out.println("Plain text after decryption is: " +
decryptPermutationCipher(encryptedPermutation, permutationKey));
                break;
            case "N":
                break;
        }
        break;

        default:
            System.out.println("Invalid choice. Please choose A or B.");
            break;
    }
}

// Handling the second Transposition Cipher choice
static void handleTranspositionCipher(Scanner scanner) {
    System.out.println("You chose Transposition Cipher.");
    System.out.println("A. Simple Transposition");
    System.out.println("B. Double Transposition");
    System.out.print("Choose A or B: ");
    String subChoice = scanner.nextLine().toUpperCase();
    String decryptionChoice;
    switch (subChoice) {
        case "A":
            System.out.println("You chose Simple Transposition.");
            System.out.print("Enter the plaintext to encrypt using Simple Transposition Cipher: ");
            String plaintextSimple = scanner.nextLine();
            System.out.print("Enter the transposition key (as comma-separated integers): ");
            String keyInputSimple = scanner.nextLine();
            int[] transpositionKey = parseKeyInput(keyInputSimple);
            String encryptedSimple = simpleTranspositionEncryptMethod(plaintextSimple, transpositionKey);
            System.out.println("Encrypted text: " + encryptedSimple);
            System.out.print("Do you want to decrypt this message?(Y/N) ");
            decryptionChoice = scanner.nextLine();
            scanner.nextLine(); // Consume newline character
            switch (decryptionChoice) {
                case "Y":
                    System.out.println("Plain text after decryption is: " +
decryptSimpleTranspositionCipher(encryptedSimple, transpositionKey));
                    break;
                case "N":
                    break;
            }
            break;

        case "B":
            System.out.println("You chose Double Transposition.");
            System.out.print("Enter the plaintext to encrypt using Double Transposition Cipher: ");
            String plaintextDouble = scanner.nextLine();
            System.out.print("Enter the first key (as comma-separated integers): ");
            int[] key1 = parseKeyInput(scanner.nextLine());

```

```

        System.out.print("Enter the second key (as comma-separated integers): ");
        int[] key2 = parseKeyInput(scanner.nextLine());
        String encryptedDouble = doubleTranspositionEncryptMethod(plaintextDouble, key1, key2);
        System.out.println("Encrypted text: " + encryptedDouble);
        System.out.print("Do you want to decrypt this message?(Y/N) ");
        decryptionChoice = scanner.nextLine();
        scanner.nextLine(); // Consume newline character
        switch (decryptionChoice) {
            case "Y":
                System.out.println("Plain text after decryption is: " +
decryptDoubleTranspositionCipher(encryptedDouble, key1, key2));
                break;
            case "N":
                break;
        }
        break;

        default:
            System.out.println("Invalid choice. Please choose A or B.");
            break;
    }
}

// Handling the third substitution Vigenere choice
static void handleVigenereCipher(Scanner scanner) {
    System.out.println("You chose Vigenere Cipher.");
    System.out.print("Enter the plaintext to encrypt using Vigenere Cipher: ");
    String plaintext = scanner.nextLine().toUpperCase();
    System.out.print("Enter the keyword: ");
    String keyword = scanner.nextLine().toUpperCase();
    String encryptedVigenere = vigenereEncryptMethod(plaintext, keyword);
    System.out.println("Encrypted Text: " + encryptedVigenere);
    System.out.print("Do you want to decrypt this message?(Y/N) ");
    String decryptionChoice = scanner.nextLine();
    scanner.nextLine(); // Consume newline character
    switch (decryptionChoice) {
        case "Y":
            System.out.println("Plain text after decryption is: " + decryptVigenere(encryptedVigenere, keyword));
            break;
        case "N":
            break;
    }
}

// Handling the fourth Encryption Algorithm choices
static void handleEncryptionAlgorithms(Scanner scanner) {
    System.out.println("You chose Encryption Algorithms.");
    System.out.println("  A. AES-128");
    System.out.println("  B. DES");
    System.out.println("  C. 3DES");
    System.out.print("Choose A, B, or C: ");
    String subChoice = scanner.nextLine().toUpperCase();
    String decryptionChoice;

    try {
        System.out.print("Enter the text you want to encrypt using the chosen algorithm: ");
        String plaintext = scanner.nextLine(); // Read user input

```

```
// Set default key and key length based on the chosen algorithm
String algorithm;
int keyLength;
String defaultKey;

switch (subChoice) {
    case "A":
        algorithm = "AES";
        keyLength = 16; // 16 characters for AES
        defaultKey = UserInput.DEFAULT_AES_KEY; // Default AES key
        break;
    case "B":
        algorithm = "DES";
        keyLength = 8; // 8 characters for DES
        defaultKey = UserInput.DEFAULT_DES_KEY; // Default DES key
        break;
    case "C":
        algorithm = "DESede"; // 3DES
        keyLength = 8; // 8 characters for 3DES
        defaultKey = UserInput.DEFAULT_3DES_KEY; // Default 3DES key
        break;
    default:
        System.out.println("Invalid choice. Please choose A, B, or C.");
        return;
}

// Option to enter encryption key or use a default key
System.out.print("Do you want to enter a custom key? (Y/N): ");
String keyChoice = scanner.nextLine().toUpperCase();
SecretKey key;

if (keyChoice.equals("Y")) {
    System.out.printf("Enter your %d-character key for %s: ", keyLength, algorithm);
    String keyInput = scanner.nextLine();
    if (keyInput.length() != keyLength) {
        System.out.printf("Key must be exactly %d characters long.\n", keyLength);
        return;
    }
    key = new SecretKeySpec(keyInput.getBytes(), algorithm);
} else {
    key = new SecretKeySpec(defaultKey.getBytes(), algorithm); // Use default key
}

// Encryption and decryption logic based on the chosen algorithm
String encryptedText;
switch (subChoice) {
    case "A": // AES-128
        encryptedText = encryptAESMethod(plaintext, key);
        System.out.println("Encrypted Text (AES-128): " + encryptedText);
        break;
    case "B": // DES
        encryptedText = encryptDESMMethod(plaintext, key);
        System.out.println("Encrypted Text (DES): " + encryptedText);
        break;
    case "C": // 3DES
        encryptedText = encrypt3DESMMethod(plaintext, key);
        break;
}
```

```

        System.out.println("Encrypted Text (3DES): " + encryptedText);
        break;
    default:
        System.out.println("Invalid choice. Please choose A, B, or C.");
        return;
    }

    // Decryption option
    System.out.print("Do you want to decrypt this message? (Y/N): ");
    decryptionChoice = scanner.nextLine().toUpperCase();
    if (decryptionChoice.equals("Y")) {
        String decryptedText = "";
        switch (subChoice) {
            case "A":
                decryptedText = decryptAES(encryptedText, key);
                break;
            case "B":
                decryptedText = decryptDES(encryptedText, key);
                break;
            case "C":
                decryptedText = decrypt3DES(encryptedText, key);
                break;
        }
        System.out.println("Plain text after decryption is: " + decryptedText);
    }

} catch (Exception e) {
    e.printStackTrace();
}
}

// Handling the fifth Encryption Mode choices
static void handleEncryptionModes(Scanner scanner) {
    System.out.println("You chose Encryption Modes.");
    System.out.println("  A. ECB");
    System.out.println("  B. CBC");
    System.out.println("  C. CFB");
    System.out.println("  D. OFB");
    System.out.print("Choose A, B, C or D: ");
    String subChoice = scanner.nextLine().toUpperCase();

    try {
        // Prompt user for plaintext
        System.out.print("Enter the plaintext to encrypt: ");
        String plaintext = scanner.nextLine(); // Read user input

        // Option to enter encryption key or use a default key
        System.out.print("Do you want to enter a custom key? (Y/N): ");
        String keyChoice = scanner.nextLine().toUpperCase();
        SecretKey key;

        if (keyChoice.equals("Y")) {
            System.out.print("Enter your 16-character key for AES: ");
            String keyInput = scanner.nextLine();
            if (keyInput.length() != 16) {
                System.out.println("Key must be exactly 16 characters long.");
            }
        }
    }
}

```



```

        return;
    }
    key = new SecretKeySpec(keyInput.getBytes(), "AES");
} else {
    key = new SecretKeySpec(UserInput.DEFAULT_AES_KEY.getBytes(), "AES"); // Use default key
}

// Generate a random initialization vector (IV)
byte[] iv = new byte[16]; // 16 bytes for AES
SecureRandom secureRandom = new SecureRandom();
secureRandom.nextBytes(iv); // Fill the IV with random bytes

String encryptedText;
String decryptionChoice;
switch (subChoice) {
    case "A":
        encryptedText = encryptAESUsingECBMethod(plaintext, key);
        System.out.println("Encrypted Text: " + encryptedText);
        System.out.print("Do you want to decrypt this message?(Y/N) ");
        decryptionChoice = scanner.nextLine();
        scanner.nextLine(); // Consume newline character
        switch (decryptionChoice) {
            case "Y":
                System.out.println("Plain text after decryption is: " + decryptAESUsingECBMode(encryptedText,
key));
                break;
            case "N":
                break;
        }
        break;

    case "B":
        encryptedText = encryptAESUsingCBCMethod(plaintext, key, iv);
        System.out.println("Encrypted Text: " + encryptedText);
        System.out.print("Do you want to decrypt this message?(Y/N) ");
        decryptionChoice = scanner.nextLine();
        scanner.nextLine(); // Consume newline character
        switch (decryptionChoice) {
            case "Y":
                System.out.println("Plain text after decryption is: " + decryptAESUsingCBC(encryptedText, key, iv));
                break;
            case "N":
                break;
        }
        break;

    case "C":
        encryptedText = encryptAESUsingCFBMethod(plaintext, key, iv);
        System.out.println("Encrypted Text: " + encryptedText);
        System.out.print("Do you want to decrypt this message?(Y/N) ");
        decryptionChoice = scanner.nextLine();
        scanner.nextLine(); // Consume newline character
        switch (decryptionChoice) {
            case "Y":
                System.out.println("Plain text after decryption is: " + decryptAESUsingCFB(encryptedText, key, iv));
                break;
            case "N":

```

```

        break;
    }
    break;

    case "D":
        encryptedText = encryptAESUsingOFBMethod(plaintext, key, iv);
        System.out.println("Encrypted Text: " + encryptedText);
        System.out.print("Do you want to decrypt this message?(Y/N) ");
        decryptionChoice = scanner.nextLine();
        scanner.nextLine(); // Consume newline character
        switch (decryptionChoice) {
            case "Y":
                System.out.println("Plain text after decryption is: " + decryptAESUsingOFB(encryptedText, key, iv));
                break;
            case "N":
                break;
        }
        break;

    default:
        System.out.println("Invalid choice. Please choose A, B, C or D.");
        return;
}

System.out.println("Encrypted Text: " + encryptedText);
} catch (Exception e) {
    e.printStackTrace();
}
}

// ***** HELPER METHODS *****

// Helper method for parsing the key input
private static int[] parseKeyInput(String keyInput) {
    String[] keyStrings = keyInput.split("");
    int[] key = new int[keyStrings.length];
    for (int i = 0; i < keyStrings.length; i++) {
        key[i] = Integer.parseInt(keyStrings[i].trim());
    }
    return key;
}

// ***** ENCRYPTION METHODS *****

// Method to encrypt plaintext using Shift Cipher
public static String shiftEncryptMethod(String text, int shift){
    // Making the String mutable
    StringBuilder ciphertext = new StringBuilder();

    // Running a for loop to iterate through characters and change their values
    // according to the requirement
    for (int i = 0; i < text.length(); i++) {
        // assigning each character consecutively to char value c
        char c = text.charAt(i);

```

```

        // Check if the character is a letter
        if (Character.isLetter(c)) {
            char base = Character.isUpperCase(c) ? 'A' : 'a';
            // Shift the letter and wrap around using modulo operation
            char shiftedChar = (char) ((c - base + shift) % 26 + base);
            ciphertext.append(shiftedChar);
        } else {
            // If not a letter, just add the character as it is
            ciphertext.append(c);
        }
    }

    return ciphertext.toString();
}

// Method to encrypt plaintext using Permutation Cipher
public static String permutationEncryptMethod(String plaintext, int[] key){
    // "HELLOWORLD" [3, 1, 4, 2]
    // Calculate the block size from the key length
    // Pad the plaintext with spaces if it isn't a multiple of the block size
    // "HELL" (first 4 characters)
    // "OWOR" (next 4 characters)
    // "LD " (padded with spaces because of missing 2 chars)
    // The letter will be given the position as mentioned in the key.
    // For eg "HELLO" -> [2, 3, 1, 4, 5] Ans: ELHLO
    // Process each block of the plaintext

    StringBuilder ciphertext = new StringBuilder();
    int blockSize = key.length;

    // Pad the plaintext with spaces if it isn't a multiple of the block size
    StringBuilder plaintextBuilder = new StringBuilder(plaintext);
    while (plaintextBuilder.length() % blockSize != 0) {
        plaintextBuilder.append(" ");
    }
    plaintext = plaintextBuilder.toString();

    // Process each block of the plaintext
    for (int i = 0; i < plaintext.length(); i += blockSize) {
        char[] block = new char[blockSize];

        // Rearrange the characters in the block according to the permutation key
        for (int j = 0; j < blockSize; j++) {
            block[j] = plaintext.charAt(i + key[j] - 1); // Apply the permutation process
        }

        // Append the rearranged block to the ciphertext
        ciphertext.append(block);
    }

    return ciphertext.toString();
}

// Method to encrypt plaintext using Simple Transposition Cipher
public static String simpleTranspositionEncryptMethod(String plaintext, int[] key){
    StringBuilder ciphertext = new StringBuilder();

```

```

// Calculate the number of rows based on key length
int numRows = (int) Math.ceil((double) plaintext.length() / key.length);

// Create a 2D array (grid) to store the plaintext in columns
char[][] grid = new char[numRows][key.length];

// Fill the grid with the plaintext characters
int index = 0;
for (int row = 0; row < numRows; row++) {
    for (int col = 0; col < key.length; col++) {
        if (index < plaintext.length()) {
            grid[row][col] = plaintext.charAt(index);
            index++;
        } else {
            // Pad with space if plaintext is shorter than the grid size
            grid[row][col] = ' ';
        }
    }
}

// Rearrange the grid according to the key and build the ciphertext
for (int i : key) {
    int keyCol = i - 1; // Key is 1-based, so we subtract 1 for 0-based indexing
    for (int row = 0; row < numRows; row++) {
        ciphertext.append(grid[row][keyCol]);
    }
}

return ciphertext.toString();
}

// Method to encrypt plaintext using Double Transposition Cipher
public static String doubleTranspositionEncryptMethod(String plaintext, int[] key1, int[] key2){
    // Perform first transposition
    String firstPass = simpleTranspositionEncryptMethod(plaintext, key1);

    // Perform second transposition on the result
    return simpleTranspositionEncryptMethod(firstPass, key2);
}

// Method to encrypt plaintext using Vigenère Cipher
public static String vigenereEncryptMethod (String plaintext, String keyword){
    StringBuilder ciphertext = new StringBuilder();
    // Make sure that the key is in uppercase, throughout
    keyword = keyword.toUpperCase();

    // Plaintext: A T T A C K A T D A W N
    // Keyword:   L E M O N L E M O N L E
    // Step 2: Shift each letter:
    // A + L = (0 + 11) = L
    // T + E = (19 + 4) = X
    // T + M = (19 + 12) = F
    // A + O = (0 + 14) = O
    // C + N = (2 + 13) = P
    // K + L = (10 + 11) = V
    // A + E = (0 + 4) = E
    // T + M = (19 + 12) = F

```

```

// D + O = (3 + 14) = R
// A + N = (0 + 13) = N
// W + L = (22 + 11) = H
// N + E = (13 + 4) = R

for (int i = 0; i < plaintext.length(); i++) {
    char plainChar = plaintext.charAt(i);

    // Formula
    // Pi = (Ci - Ki) + 26
    // Pi - decrypted plaintext letter.
    // Ci - ciphertext letter (converted to an index 0–25).
    // Ki - keyword letter (converted to an index 0–25).
    // Adding 26 ensures we avoid negative numbers during the subtraction.

    if (Character.isLetter(plainChar)) {
        int shift = (plainChar - 'A' + keyword.charAt(i % keyword.length()) - 'A') % 26;
        ciphertext.append((char) (shift + 'A'));
    } else {
        ciphertext.append(plainChar); // Non-letter characters remain unchanged
    }
}

return ciphertext.toString();
}

// Method to encrypt plaintext using AES Algorithm
// We implement it using ECB by default
public static String encryptAESMethod(String plaintext, SecretKey key) throws Exception {
    // specifies the encryption algorithm (AES), the mode of operation (ECB), and the padding scheme (PKCS5).
    // AES: Advanced Encryption Standard, a symmetric key algorithm.
    // ECB (Electronic Codebook): A mode of operation that divides plaintext into blocks and encrypts each block
    // independently.
    // PKCS5Padding: A padding scheme that ensures that the plaintext is a multiple of the block size (16 bytes
    // for AES).
    Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding"); // AES in ECB mode with padding
    // Initialises cipher instance in encryption mode using the provided secret key.
    // Key is used to set up the cipher's internal state for encryption.
    // This key must be the same during decryption to retrieve the original plaintext.
    cipher.init(Cipher.ENCRYPT_MODE, key); // Initialize cipher for encryption
    // plaintext.getBytes() converts the input string into a byte array using the default character encoding.
    // cipher.doFinal() method processes the input bytes, encrypting them based on the initialized cipher.
    // This method returns an array of bytes representing the encrypted data.
    byte[] encryptedBytes = cipher.doFinal(plaintext.getBytes()); // Encrypt the plaintext
    // Converts the encrypted byte array into a Base64 encoded string.
    return Base64.getEncoder().encodeToString(encryptedBytes); // Convert encrypted bytes to Base64 string
}

// Method to encrypt plaintext using DES Algorithm
public static String encryptDESMMethod(String plaintext, SecretKey key) throws Exception {
    // Create a Cipher object for DES
    // Code is similar to AES
    Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
    cipher.init(Cipher.ENCRYPT_MODE, key); // Initialize the cipher for encryption
    byte[] encryptedBytes = cipher.doFinal(plaintext.getBytes()); // Encrypt the plaintext
    return Base64.getEncoder().encodeToString(encryptedBytes); // Convert to Base64 string
}

```

```

// Method to encrypt plaintext using 3DES Algorithm
public static String encrypt3DESMethod(String plaintext, SecretKey key) throws Exception {
    // Create a Cipher object for 3DES
    // Code is similar to AES
    Cipher cipher = Cipher.getInstance("DESede/ECB/PKCS5Padding");
    cipher.init(Cipher.ENCRYPT_MODE, key); // Initialize the cipher for encryption

    byte[] encryptedBytes = cipher.doFinal(plaintext.getBytes()); // Encrypt the plaintext
    return Base64.getEncoder().encodeToString(encryptedBytes); // Convert to Base64 string
}

// Method to encrypt plaintext using AES in ECB mode
public static String encryptAESUsingECBMethod(String plaintext, SecretKey key) throws Exception {
    Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding"); // AES in ECB mode with padding
    cipher.init(Cipher.ENCRYPT_MODE, key); // Initialize the cipher for encryption

    byte[] encryptedBytes = cipher.doFinal(plaintext.getBytes()); // Encrypt the plaintext
    return Base64.getEncoder().encodeToString(encryptedBytes); // Convert to Base64 string
}

// Method to encrypt plaintext using AES in CBC mode
public static String encryptAESUsingCBCMethod(String plaintext, SecretKey key, byte[] iv) throws Exception {
    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding"); // AES in CBC mode with padding
    IvParameterSpec ivParams = new IvParameterSpec(iv); // Create IV parameter spec
    cipher.init(Cipher.ENCRYPT_MODE, key, ivParams); // Initialize the cipher for encryption

    byte[] encryptedBytes = cipher.doFinal(plaintext.getBytes()); // Encrypt the plaintext
    return Base64.getEncoder().encodeToString(encryptedBytes); // Convert to Base64 string
}

// Method to encrypt plaintext using AES in CFB mode
public static String encryptAESUsingCFBMethod(String plaintext, SecretKey key, byte[] iv) throws Exception {
    Cipher cipher = Cipher.getInstance("AES/CFB/PKCS5Padding"); // AES in CFB mode with padding
    IvParameterSpec ivParams = new IvParameterSpec(iv); // Create IV parameter spec
    cipher.init(Cipher.ENCRYPT_MODE, key, ivParams); // Initialize the cipher for encryption

    byte[] encryptedBytes = cipher.doFinal(plaintext.getBytes()); // Encrypt the plaintext
    return Base64.getEncoder().encodeToString(encryptedBytes); // Convert to Base64 string
}

// Method to encrypt plaintext using AES in OFB mode
public static String encryptAESUsingOFBMethod(String plaintext, SecretKey key, byte[] iv) throws Exception {
    Cipher cipher = Cipher.getInstance("AES/OFB/PKCS5Padding"); // AES in OFB mode with padding
    IvParameterSpec ivParams = new IvParameterSpec(iv); // Create IV parameter spec
    cipher.init(Cipher.ENCRYPT_MODE, key, ivParams); // Initialize the cipher for encryption

    byte[] encryptedBytes = cipher.doFinal(plaintext.getBytes()); // Encrypt the plaintext
    return Base64.getEncoder().encodeToString(encryptedBytes); // Convert to Base64 string
}

// ***** DECRYPTION METHODS *****

// Method to decrypt ciphertext using Shift Cipher
public static String decryptShiftCipher(String encryptedText, int shift) {
    StringBuilder decryptedText = new StringBuilder();

```

```

// Iterate through each character in the encrypted text
for (char ch : encryptedText.toCharArray()) {
    // Check if the character is an uppercase letter
    if (Character.isUpperCase(ch)) {
        char decryptedChar = (char) (((ch - 'A' - shift + 26) % 26) + 'A'); // Adjust with modulo
        decryptedText.append(decryptedChar);
    }
    // Check if the character is a lowercase letter
    else if (Character.isLowerCase(ch)) {
        char decryptedChar = (char) (((ch - 'a' - shift + 26) % 26) + 'a'); // Adjust with modulo
        decryptedText.append(decryptedChar);
    } else {
        // If it's not a letter, just append it unchanged
        decryptedText.append(ch);
    }
}

return decryptedText.toString(); // Return the decrypted text
}

// Method to decrypt ciphertext using Permutation Cipher
public static String decryptPermutationCipher(String ciphertext, int[] key) {
    // Create a char array for the decrypted text
    StringBuilder plaintext = new StringBuilder();
    int blockSize = key.length;

    // Process each block of the ciphertext
    for (int i = 0; i < ciphertext.length(); i += blockSize) {
        char[] block = new char[blockSize];

        // Rearrange the characters in the block back to their original positions using the inverse of the permutation
        key
        for (int j = 0; j < blockSize; j++) {
            block[key[j] - 1] = ciphertext.charAt(i + j); // Reverse the permutation process
        }

        // Append the rearranged block to the plaintext
        plaintext.append(block);
    }

    return plaintext.toString();
}

// Method to decrypt ciphertext using Single Transposition
public static String decryptSimpleTranspositionCipher(String encryptedText, int[] key) {
    int numCols = key.length;
    int numRows = (int) Math.ceil((double) encryptedText.length() / numCols);
    char[][] grid = new char[numRows][numCols];

    // Fill the grid column by column according to the key
    int index = 0;
    for (int j : key) {
        int colIndex = j - 1; // Convert to 0-based index
        for (int row = 0; row < numRows; row++) {
            if (index < encryptedText.length()) {
                grid[row][colIndex] = encryptedText.charAt(index);
                index++;
            }
        }
    }

```

```

        } else {
            // Pad with spaces if necessary
            grid[row][colIndex] = ' ';
        }
    }
}

// Build the decrypted text by reading the grid row-wise
StringBuilder decryptedText = new StringBuilder();
for (int row = 0; row < numRows; row++) {
    for (int col = 0; col < numCols; col++) {
        decryptedText.append(grid[row][col]);
    }
}

return decryptedText.toString().trim(); // Trim any trailing spaces
}

// Method to decrypt ciphertext using Double Transposition
public static String decryptDoubleTranspositionCipher(String encryptedText, int[] key1, int[] key2) {
    // First, decrypt using the second key (reverse order)
    String intermediateText = decryptSimpleTranspositionCipher(encryptedText, key2);
    // Then, decrypt the intermediate text using the first key (reverse order)
    return decryptSimpleTranspositionCipher(intermediateText, key1);
}

// Method to decrypt ciphertext using AES
public static String decryptAES(String encryptedText, SecretKey key) throws Exception {
    Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding"); // AES in ECB mode with padding
    cipher.init(Cipher.DECRYPT_MODE, key); // Initialize cipher for decryption
    byte[] decryptedBytes = cipher.doFinal(Base64.getDecoder().decode(encryptedText)); // Decrypt the
ciphertext
    return new String(decryptedBytes); // Convert decrypted bytes to string
}

// Method to decrypt ciphertext using DES
public static String decryptDES(String ciphertext, SecretKey key) throws Exception {
    Cipher cipher = Cipher.getInstance("DES/ECB/PKCS5Padding");
    cipher.init(Cipher.DECRYPT_MODE, key); // Initialize the cipher for decryption

    byte[] decryptedBytes = cipher.doFinal(Base64.getDecoder().decode(ciphertext)); // Decrypt the Base64
string
    return new String(decryptedBytes); // Convert decrypted bytes back to string
}

// Method to decrypt ciphertext using 3DES
public static String decrypt3DES(String ciphertext, SecretKey key) throws Exception {
    Cipher cipher = Cipher.getInstance("DESede/ECB/PKCS5Padding");
    cipher.init(Cipher.DECRYPT_MODE, key); // Initialize the cipher for decryption

    byte[] decryptedBytes = cipher.doFinal(Base64.getDecoder().decode(ciphertext)); // Decrypt the Base64
string
    return new String(decryptedBytes); // Convert decrypted bytes back to string
}

// Decrypt the ciphertext using the Vigenère Cipher
public static String decryptVigenere(String ciphertext, String keyword) {

```



```

StringBuilder plaintext = new StringBuilder();
keyword = keyword.toUpperCase();

for (int i = 0; i < ciphertext.length(); i++) {
    char cipherChar = ciphertext.charAt(i);

    if (Character.isLetter(cipherChar)) {
        int shift = (cipherChar - keyword.charAt(i % keyword.length()) + 26) % 26;
        plaintext.append((char) (shift + 'A'));
    } else {
        plaintext.append(cipherChar); // Non-letter characters remain unchanged
    }
}

return plaintext.toString();
}

// Method to decrypt ciphertext using AES in ECB mode
public static String decryptAESUsingECBMode(String ciphertext, SecretKey key) throws Exception {
    Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding"); // Create a Cipher object for AES
    cipher.init(Cipher.DECRYPT_MODE, key); // Initialize the cipher for decryption

    byte[] decryptedBytes = cipher.doFinal(Base64.getDecoder().decode(ciphertext)); // Decrypt the Base64
string
    return new String(decryptedBytes); // Convert decrypted bytes back to string
}

// Method to decrypt ciphertext using AES in CBC mode
public static String decryptAESUsingCBC(String ciphertext, SecretKey key, byte[] iv) throws Exception {
    Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding"); // Create a Cipher object for AES
    IvParameterSpec ivParams = new IvParameterSpec(iv); // Create IV parameter spec
    cipher.init(Cipher.DECRYPT_MODE, key, ivParams); // Initialize the cipher for decryption

    byte[] decryptedBytes = cipher.doFinal(Base64.getDecoder().decode(ciphertext)); // Decrypt the Base64
string
    return new String(decryptedBytes); // Convert decrypted bytes back to string
}

// Method to decrypt ciphertext using AES in CFB mode
public static String decryptAESUsingCFB(String ciphertext, SecretKey key, byte[] iv) throws Exception {
    Cipher cipher = Cipher.getInstance("AES/CFB/PKCS5Padding"); // Create a Cipher object for AES
    IvParameterSpec ivParams = new IvParameterSpec(iv); // Create IV parameter spec
    cipher.init(Cipher.DECRYPT_MODE, key, ivParams); // Initialize the cipher for decryption

    byte[] decryptedBytes = cipher.doFinal(Base64.getDecoder().decode(ciphertext)); // Decrypt the Base64
string
    return new String(decryptedBytes); // Convert decrypted bytes back to string
}

// Method to decrypt ciphertext using AES in OFB mode
public static String decryptAESUsingOFB(String ciphertext, SecretKey key, byte[] iv) throws Exception {
    Cipher cipher = Cipher.getInstance("AES/OFB/PKCS5Padding"); // Create a Cipher object for AES
    IvParameterSpec ivParams = new IvParameterSpec(iv); // Create IV parameter spec
    cipher.init(Cipher.DECRYPT_MODE, key, ivParams); // Initialize the cipher for decryption

    byte[] decryptedBytes = cipher.doFinal(Base64.getDecoder().decode(ciphertext)); // Decrypt the Base64
string

```

```

    return new String(decryptedBytes); // Convert decrypted bytes back to string
}
}

```

### UserInput.java

```

import java.util.Scanner;

public class UserInput {

    public static final String DEFAULT_AES_KEY = "DEFAULTAESKEY123"; // 16 characters for AES
    public static final String DEFAULT_DES_KEY = "DEFAULTK"; // 8 characters for DES
    public static final String DEFAULT_3DES_KEY = "DEFAULTDESKEY12345678901";

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);
        int choice = 0;
        boolean exit = false;

        while (!exit) {
            // Display the main menu
            System.out.println("=====");
            System.out.println("                        Menu                        ");
            System.out.println("=====");
            System.out.println(" 1. Substitution Cipher");
            System.out.println("   A. Shift Cipher");
            // HELLO 3 -> KHOOR
            System.out.println("   B. Permutation Cipher");
            // HELLO 3,1,4,2,5 -> ELHLO
            System.out.println(" 2. Transposition Cipher");
            System.out.println("   A. Simple Transposition");
            // HELLO WORLD 2,1,4,3 -> e lHorlo lwd
            System.out.println("   B. Double Transposition");
            // HELLO WORLD 3,1,4,2 2,1,3,4 -> hloelw orld o
            System.out.println(" 3. Vigenere Cipher");
            // HELLO KEY
            System.out.println(" 4. Encryption Algorithms");
            System.out.println("   A. AES-128");
            System.out.println("   B. DES");
            System.out.println("   C. 3DES");
            System.out.println(" 5. Encryption Modes");
            System.out.println("   A. ECB");
            System.out.println("   B. CBC");
            System.out.println("   C. CFB");
            System.out.println("   D. OFB");
            System.out.println(" 6. Exit");
            System.out.println("=====");
            // Prompt for user choice
            System.out.print("Enter your choice (1-6): ");
            try {
                choice = scanner.nextInt();
                scanner.nextLine(); // Consume the newline character
            }
        }
    }
}

```

```
// Check if the choice is within the valid range
if (choice < 1 || choice > 6) {
    System.out.println("Invalid choice. Please select a number between 1 and 6.");
    continue; // Re-prompt for input
}
switch (choice) {
    case 1:
        Functionalities.handleSubstitutionCipher(scanner);
        break;
    case 2:
        Functionalities.handleTranspositionCipher(scanner);
        break;
    case 3:
        Functionalities.handleVigenereCipher(scanner);
        break;
    case 4:
        Functionalities.handleEncryptionAlgorithms(scanner);
        break;
    case 5:
        Functionalities.handleEncryptionModes(scanner);
        break;
    case 6:
        System.out.println("Exiting...");
        exit = true;
        break;
    default:
        System.out.println("Invalid choice. Please choose between 1 and 6.");
        break;
}
} catch (Exception e) {
    throw new RuntimeException(e);
}

System.out.println();
}

scanner.close();
}

}
```

**\*\* END OF DOCUMENT \*\***