Below is an example "lab manual" style document that includes the **aim**, **procedure**, **algorithm description**, and **simple source code** (in Python) for both encryption and decryption for several cipher experiments. You can use these examples for your experiments on:

1. **Caesar Cipher**
2. **Substitution Cipher**
3. **Vigenère Cipher**
4. **Hill Cipher**
5. **Vernam Cipher (One-Time Pad)**
6. **Columnar Transposition Cipher**
7. **DES (Data Encryption Standard)**
8. **AES (Advanced Encryption Standard)**

---

**1. Caesar Cipher**

**Aim**

Implement a simple shift cipher where each letter in the plaintext is shifted by a fixed number (the key) along the alphabet.

**Procedure**

1. Choose a shift key (an integer).
2. For each letter in the plaintext:
   - Convert it to its alphabetical index.
   - **Encryption:** Add the key (using modulo 26 to wrap around).
   - **Decryption:** Subtract the key (again modulo 26).
3. Convert the numeric value back to a letter.

**Algorithm**

- **Encryption:**
  For a letter *L*:
  E(L) = ( (ord(L) - base) + key ) mod 26 + base
  where *base* is ord('A') (or ord('a') for lowercase).
- **Decryption:**
  For a letter *C*:
  D(C) = ( (ord(C) - base) - key ) mod 26 + base

**Python Source Code**

```
def caesar_encrypt(plaintext, key):

    result = ""
```

```
    for char in plaintext:
        if char.isalpha():
            shift = key % 26
            base = ord('A') if char.isupper() else ord('a')
            result += chr((ord(char) - base + shift) % 26 + base)
        else:
            result += char
    return result

def caesar_decrypt(ciphertext, key):
    # Decryption is just encryption with negative key
    return caesar_encrypt(ciphertext, -key)

# Example usage:
plain = "HELLO WORLD"
key = 3
encrypted = caesar_encrypt(plain, key)
decrypted = caesar_decrypt(encrypted, key)
print("Caesar Cipher:")
print("Plaintext: ", plain)
print("Encrypted: ", encrypted)
print("Decrypted: ", decrypted)
```

**Experiment Problem**

Encrypt and decrypt the message **"HELLO WORLD"** with a shift key of **3**. Verify that decryption returns the original text.

---

**2. Substitution Cipher**

**Aim**

Implement a substitution cipher where each letter in the plaintext is replaced by another letter using a predetermined permutation (mapping) of the alphabet.

**Procedure**

1. Define a substitution mapping (e.g., A → Q, B → W, …).

2. For encryption, replace each plaintext letter with its corresponding letter from the mapping.
3. For decryption, use the inverse mapping.

**Algorithm**

- **Encryption:**
  For each letter *L* in the plaintext, replace it with *mapping[L]*.
- **Decryption:**
  Create the inverse mapping (i.e. for each pair (k, v), set inverse_mapping[v] = k) and then replace each ciphertext letter with its corresponding plaintext letter.

**Python Source Code**

```
def substitution_encrypt(plaintext, mapping):
    ciphertext = ""
    for char in plaintext:
        if char.upper() in mapping:
            new_char = mapping[char.upper()]
            ciphertext += new_char if char.isupper() else new_char.lower()
        else:
            ciphertext += char
    return ciphertext


def substitution_decrypt(ciphertext, mapping):
    inverse_mapping = {v: k for k, v in mapping.items()}
    plaintext = ""
    for char in ciphertext:
        if char.upper() in inverse_mapping:
            new_char = inverse_mapping[char.upper()]
            plaintext += new_char if char.isupper() else new_char.lower()
        else:
            plaintext += char
    return plaintext


# Example mapping (A simple fixed mapping)
mapping = {
```

```
    'A': 'Q', 'B': 'W', 'C': 'E', 'D': 'R', 'E': 'T', 'F': 'Y', 'G': 'U',
    'H': 'I', 'I': 'O', 'J': 'P', 'K': 'A', 'L': 'S', 'M': 'D', 'N': 'F',
    'O': 'G', 'P': 'H', 'Q': 'J', 'R': 'K', 'S': 'L', 'T': 'Z', 'U': 'X',
    'V': 'C', 'W': 'V', 'X': 'B', 'Y': 'N', 'Z': 'M'
}

plain = "HELLO WORLD"
encrypted = substitution_encrypt(plain, mapping)
decrypted = substitution_decrypt(encrypted, mapping)
print("\nSubstitution Cipher:")
print("Plaintext: ", plain)
print("Encrypted: ", encrypted)
print("Decrypted: ", decrypted)
```

**Experiment Problem**

Using the above mapping, encrypt and decrypt **"HELLO WORLD"**. Check that the decryption correctly restores the original text.

---

**3. Vigenère Cipher**

**Aim**

Implement the Vigenère cipher, which uses a keyword to perform a series of Caesar cipher shifts on the plaintext.

**Procedure**

1. Choose a keyword.
2. For encryption, use the corresponding letter of the key (repeating cyclically) to determine the shift for each plaintext letter.
3. For decryption, subtract the key’s shift.

**Algorithm**

For each character at position *i*:

- **Encryption:**
  C_i = (P_i + K_j) mod 26
- **Decryption:**
  P_i = (C_i - K_j) mod 26

where $K_j$ is the numerical value (A=0, B=1, …) of the key letter at position *(i mod key_length)*.

**Python Source Code**

```
def vigenere_encrypt(plaintext, key):
    ciphertext = ""
    key = key.upper()
    key_length = len(key)
    for i, char in enumerate(plaintext):
        if char.isalpha():
            base = ord('A') if char.isupper() else ord('a')
            k = ord(key[i % key_length]) - ord('A')
            shifted = (ord(char) - base + k) % 26
            ciphertext += chr(shifted + base)
        else:
            ciphertext += char
    return ciphertext


def vigenere_decrypt(ciphertext, key):
    plaintext = ""
    key = key.upper()
    key_length = len(key)
    for i, char in enumerate(ciphertext):
        if char.isalpha():
            base = ord('A') if char.isupper() else ord('a')
            k = ord(key[i % key_length]) - ord('A')
            shifted = (ord(char) - base - k) % 26
            plaintext += chr(shifted + base)
        else:
            plaintext += char
    return plaintext
```

```
plain = "HELLO WORLD"

key = "KEY"

encrypted = vigenere_encrypt(plain, key)

decrypted = vigenere_decrypt(encrypted, key)

print("\nVigenère Cipher:")

print("Plaintext: ", plain)

print("Encrypted: ", encrypted)

print("Decrypted: ", decrypted)
```

**Experiment Problem**

Encrypt and decrypt **"HELLO WORLD"** using the key **"KEY"**. Confirm that decryption reproduces the original plaintext.

---

**4. Hill Cipher**

**Aim**

Implement the Hill cipher that uses matrix multiplication over modulo arithmetic to encrypt blocks of text.

**Procedure**

1. Choose an invertible key matrix (commonly 2×2 for simplicity) modulo 26.
2. Convert the plaintext into numerical vectors (A=0, B=1, …).
3. Divide the plaintext into blocks matching the matrix size (pad if needed).
4. **Encryption:** Multiply each block by the key matrix modulo 26.
5. **Decryption:** Compute the inverse key matrix modulo 26 and multiply each ciphertext block by it.

**Algorithm**

- **Encryption:**
  For block vector *P*:
  $C = (K \times P) \bmod 26$
- **Decryption:**
  $P = (K\_inv \times C) \bmod 26$
  where *K_inv* is the modular inverse of *K*.

**Python Source Code**

```
import numpy as np
```

```
def mod_inverse_matrix(matrix, modulus):
    # Only supports 2x2 matrices for simplicity.
    det = int(round(np.linalg.det(matrix))) % modulus
    det_inv = None
    for i in range(1, modulus):
        if (det * i) % modulus == 1:
            det_inv = i
            break
    if det_inv is None:
        raise ValueError("Matrix is not invertible modulo", modulus)
    # Compute adjugate matrix for 2x2:
    inv_matrix = np.array([[matrix[1,1], -matrix[0,1]],
                           [-matrix[1,0], matrix[0,0]]])
    inv_matrix = (det_inv * inv_matrix) % modulus
    return inv_matrix.astype(int)


def text_to_numbers(text):
    return [ord(char) - ord('A') for char in text]


def numbers_to_text(numbers):
    return ''.join(chr(num + ord('A')) for num in numbers)


def hill_encrypt(plaintext, key_matrix):
    n = key_matrix.shape[0]
    plaintext = plaintext.replace(" ", "")
    if len(plaintext) % n != 0:
        plaintext += 'X' * (n - len(plaintext) % n)
    ciphertext = ""
    for i in range(0, len(plaintext), n):
        block = plaintext[i:i+n]
        block_nums = np.array(text_to_numbers(block))
```

```python
        cipher_block = np.dot(key_matrix, block_nums) % 26
        ciphertext += numbers_to_text(cipher_block)
    return ciphertext

def hill_decrypt(ciphertext, key_matrix):
    n = key_matrix.shape[0]
    inv_key = mod_inverse_matrix(key_matrix, 26)
    plaintext = ""
    for i in range(0, len(ciphertext), n):
        block = ciphertext[i:i+n]
        block_nums = np.array(text_to_numbers(block))
        plain_block = np.dot(inv_key, block_nums) % 26
        plaintext += numbers_to_text(plain_block)
    return plaintext

# Example key matrix (2x2) – must be invertible modulo 26.
key_matrix = np.array([[3, 3],
                       [2, 5]])
plain = "HELLO"
encrypted = hill_encrypt(plain, key_matrix)
decrypted = hill_decrypt(encrypted, key_matrix)
print("\nHill Cipher:")
print("Plaintext: ", plain)
print("Encrypted: ", encrypted)
print("Decrypted: ", decrypted)
```

**Experiment Problem**

Choose a 2×2 key matrix (for example, [[3, 3], [2, 5]]) and use it to encrypt and decrypt **"HELLO"**. Verify that decryption returns the original message.

---

**5. Vernam Cipher (One-Time Pad)**

**Aim**

Implement the Vernam cipher (a one-time pad) where the plaintext is XORed with a random key of equal length, offering perfect secrecy when used correctly.

**Procedure**

1. Generate a random key of the same length as the plaintext.
2. Convert the plaintext to bytes.
3. **Encryption:** XOR each byte of plaintext with the corresponding key byte.
4. **Decryption:** XOR the ciphertext with the same key to recover the plaintext.

**Algorithm**

- **Encryption/Decryption:**
  For each byte *b*:
  result_byte = plaintext_byte XOR key_byte

**Python Source Code**

```
import os


def vernam_encrypt(plaintext, key):
    plaintext_bytes = plaintext.encode('utf-8')
    ciphertext = bytes([b ^ k for b, k in zip(plaintext_bytes, key)])
    return ciphertext


def vernam_decrypt(ciphertext, key):
    plaintext_bytes = bytes([b ^ k for b, k in zip(ciphertext, key)])
    return plaintext_bytes.decode('utf-8')


plain = "HELLO"
key = os.urandom(len(plain))  # Random key of the same length as plaintext
encrypted = vernam_encrypt(plain, key)
decrypted = vernam_decrypt(encrypted, key)
print("\nVernam Cipher (One-Time Pad):")
print("Plaintext: ", plain)
print("Encrypted (hex): ", encrypted.hex())
print("Decrypted: ", decrypted)
```

**Experiment Problem**

Generate a random key for **"HELLO"**, encrypt the text using the Vernam cipher, and then decrypt it. Check that the decrypted text is identical to the original plaintext.

---

**6. Columnar Transposition Cipher**

**Aim**

Implement the columnar transposition cipher that rearranges the plaintext letters by writing them in a grid and then reading the columns in a specified order based on a key.

**Procedure**

1. Choose a key (which defines the number and order of columns).
2. Write the plaintext (without spaces) row-wise in a grid.
3. **Encryption:** Rearrange columns according to the sorted order of the key's characters and read off column-wise.
4. **Decryption:** Reverse the process by filling the columns in sorted order and then reading the grid row-wise.

**Algorithm**

- **Encryption:**
  1. Remove spaces and pad the plaintext if needed.
  2. Write into a matrix with columns equal to the key length.
  3. Sort the key characters; for each sorted index, read the column.
- **Decryption:**
  1. Determine the number of rows.
  2. Fill the matrix column-wise in the order defined by the sorted key.
  3. Read the matrix row-wise.

**Python Source Code**

```
def columnar_encrypt(plaintext, key):
    plaintext = plaintext.replace(" ", "")
    num_cols = len(key)
    num_rows = (len(plaintext) + num_cols - 1) // num_cols
    padded = plaintext.ljust(num_rows * num_cols, 'X')

    matrix = [list(padded[i*num_cols:(i+1)*num_cols]) for i in range(num_rows)]
    key_order = sorted(list(enumerate(key)), key=lambda x: x[1])
```

```
    ciphertext = ""
    for col_index, _ in key_order:
        for row in matrix:
            ciphertext += row[col_index]
    return ciphertext

def columnar_decrypt(ciphertext, key):
    num_cols = len(key)
    num_rows = (len(ciphertext) + num_cols - 1) // num_cols
    key_order = sorted(list(enumerate(key)), key=lambda x: x[1])
    matrix = [[''] * num_cols for _ in range(num_rows)]
    index = 0
    for col_index, _ in key_order:
        for row in range(num_rows):
            if index < len(ciphertext):
                matrix[row][col_index] = ciphertext[index]
                index += 1
    plaintext = "".join("".join(row) for row in matrix)
    return plaintext.rstrip('X')

plain = "HELLO WORLD"
key = "4312567"  # Example key (digits can represent column order)
encrypted = columnar_encrypt(plain, key)
decrypted = columnar_decrypt(encrypted, key)
print("\nColumnar Transposition Cipher:")
print("Plaintext: ", plain)
print("Encrypted: ", encrypted)
print("Decrypted: ", decrypted)
```

**Experiment Problem**

Encrypt and decrypt **"HELLO WORLD"** using a columnar transposition cipher with the key **"4312567"**. Confirm that decryption restores the original message.

## 7. DES (Data Encryption Standard)

### Aim

Use the DES algorithm (a symmetric-key block cipher) to encrypt and decrypt data with a fixed key size of 8 bytes.

### Procedure

1. Install/import a cryptographic library (e.g., PyCryptodome).
2. Create a DES cipher object (using a mode such as ECB).
3. Pad the plaintext to a multiple of DES's block size (8 bytes).
4. Encrypt the padded plaintext.
5. Decrypt and then remove the padding.

### Algorithm

- **Encryption:**
  Use DES from the library after padding the plaintext.
- **Decryption:**
  Decrypt the ciphertext and then unpad to recover the plaintext.

### Python Source Code

```
from Crypto.Cipher import DES

from Crypto.Util.Padding import pad, unpad


def des_encrypt(plaintext, key):

    cipher = DES.new(key, DES.MODE_ECB)

    padded_text = pad(plaintext.encode(), DES.block_size)

    ciphertext = cipher.encrypt(padded_text)

    return ciphertext


def des_decrypt(ciphertext, key):

    cipher = DES.new(key, DES.MODE_ECB)

    padded_text = cipher.decrypt(ciphertext)

    plaintext = unpad(padded_text, DES.block_size)

    return plaintext.decode()
```

```
plain = "HELLO DES"

key_des = b'8bytekey'  # Key must be exactly 8 bytes

encrypted = des_encrypt(plain, key_des)

decrypted = des_decrypt(encrypted, key_des)

print("\nDES Cipher:")

print("Plaintext: ", plain)

print("Encrypted (hex): ", encrypted.hex())

print("Decrypted: ", decrypted)
```

**Experiment Problem**

Using an 8-byte key (e.g., **"8bytekey"**), encrypt and decrypt **"HELLO DES"** in DES ECB mode. Verify that the decrypted text matches the original plaintext.

---

**8. AES (Advanced Encryption Standard)**

**Aim**

Use the AES algorithm to encrypt and decrypt data securely using a 16-byte key (for AES-128).

**Procedure**

1. Import the necessary modules from a cryptographic library (e.g., PyCryptodome).
2. Create an AES cipher object in CBC mode with a random Initialization Vector (IV).
3. Pad the plaintext to AES's block size (16 bytes).
4. Encrypt the plaintext (prepend IV for later decryption).
5. Decrypt by extracting the IV and unpadding the result.

**Algorithm**

- **Encryption:**
  ciphertext = IV || AES.new(key, AES.MODE_CBC, IV).encrypt(pad(plaintext))
- **Decryption:**
  Extract IV, decrypt the ciphertext, and then unpad.

**Python Source Code**

```
from Crypto.Cipher import AES

from Crypto.Util.Padding import pad, unpad

import os


def aes_encrypt(plaintext, key):
```

```python
    iv = os.urandom(AES.block_size)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    ciphertext = cipher.encrypt(pad(plaintext.encode(), AES.block_size))
    return iv + ciphertext  # Prepend IV for use in decryption


def aes_decrypt(ciphertext, key):
    iv = ciphertext[:AES.block_size]
    actual_ciphertext = ciphertext[AES.block_size:]
    cipher = AES.new(key, AES.MODE_CBC, iv)
    plaintext = unpad(cipher.decrypt(actual_ciphertext), AES.block_size)
    return plaintext.decode()


plain = "HELLO AES"
key_aes = b'16bytekeyforaes!'  # Must be 16 bytes for AES-128
encrypted = aes_encrypt(plain, key_aes)
decrypted = aes_decrypt(encrypted, key_aes)
print("\nAES Cipher:")
print("Plaintext: ", plain)
print("Encrypted (hex): ", encrypted.hex())
print("Decrypted: ", decrypted)
```

**Experiment Problem**

Encrypt and decrypt **"HELLO AES"** using AES in CBC mode with a 16-byte key (e.g., **"16bytekeyforaes!"**). Verify that the decrypted text is identical to the original plaintext.

---

Each of these experiments demonstrates a different method of encryption and decryption—from classical ciphers like Caesar and Vigenère to modern symmetric-key methods like DES and AES. You can modify the keys, plaintexts, or even extend the algorithms (for example, by handling different block sizes or modes) to further explore the concepts.