Source Code of Caesar Cipher:

```
def encrypt_decrypt(text, mode, key):
    result = ''
    if mode == 'd':
        key = -key
    else:
        key = key

    for letter in text:
        letter = letter.lower()
        if not letter == ' ':
            index = letters.find(letter)
            if index == -1:
                result += letter
            else:
                new_index = index + key
                if new_index >= num_letters:
                    new_index -= num_letters
                elif new_index < 0:
                    new_index += num_letters
                result += letters[new_index]
    return result


# input section
letters = 'abcdefghijklmnopqrstuvwxyz'
plaintext = input('Enter your text : ')
mode = input("e/d: ")
num_letters = len(letters)
key = int(input('Enter key through 1 to 26 : '))
value = encrypt_decrypt(plaintext, mode, key)
print(value)
```

Output:

Encryption

Enter your text: hello3

e/d: e

Enter key through 1 to 26 : 3

khoor3

Decryption

Enter your text : khoor3

e/d: d

Enter key through 1 to 26 : 3

hello3

Source code of Mono-Alphabetic cipher:

```
class MonoalphabeticCipher:
    def __init__(self):
        self.normal_char = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
                            'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r',
                            's', 't', 'u', 'v', 'w', 'x', 'y', 'z']

        self.coded_char = ['Q', 'W', 'E', 'R', 'T', 'Y', 'U', 'I', 'O',
                           'P', 'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K',
                           'L', 'Z', 'X', 'C', 'V', 'B', 'N', 'M']

    def string_encryption(self, s):
        encrypted_string = ""
        for char in s:
            for i in range(26):
                if char == self.normal_char[i]:
                    encrypted_string += self.coded_char[i]
                    break
                elif char < 'a' or char > 'z':
                    encrypted_string += char
                    break
        return encrypted_string

    def string_decryption(self, s):
        decrypted_string = ""
        for char in s:
            for i in range(26):
                if char == self.coded_char[i]:
                    decrypted_string += self.normal_char[i]
                    break
                elif char < 'A' or char > 'Z':
                    decrypted_string += char
                    break
        return decrypted_string
```

Output:

Plain text: I am ICEIAN

Encrypted message: O QD OETOQF

Decrypted message: i am iceian

Source Code for Playfair cipher:

```
import string

def key_matrix_generation(key):
    atoz = string.ascii_lowercase.replace('j', '.')

    key_matrix = ['' for i in range(5)]

    i = 0
    j = 0

    for c in key:
        if c in atoz:
            key_matrix[i] += c
            atoz = atoz.replace(c, '.')

            j += 1
            if j > 4:
                i += 1
                j = 0

    for c in atoz:
        if c != '.':
            key_matrix[i] += c

            j += 1
            if j > 4:
                i += 1
                j = 0

    return key_matrix


key_matrix = key_matrix_generation('playfire example')


def encryption(plaintext, key_matrix):
    plaintextpair = []
    ciphertextpairs = []
    i = 0
```

```
while i < len(plaintext):
    a = plaintext[i]
    b = ''

    if i + 1 == len(plaintext):
        b += 'x'
    else:
        b = plaintext[i + 1]

    if a != b:
        plaintextpair.append(a + b)
        i += 2
    else:
        plaintextpair.append(a + 'x')
        i += 1

# rule 2

for pair in plaintextpair:
    applied_rule = False
    for row in key_matrix:
        if pair[0] in row and pair[1] in row:
            j0 = row.find(pair[0])
            j1 = row.find(pair[1])

            ciphertextpair = row[(j0 + 1) % 5] + row[(j1 + 1) % 5]
            ciphertextpairs.append(ciphertextpair)
            applied_rule = True

    if applied_rule:
        continue

    for j in range(5):
        col = "".join(key_matrix[i][j] for i in range(5))
        if pair[0] in col and pair[1] in col:
            i0 = col.find(pair[0])
            i1 = col.find(pair[1])

            ciphertextpair = col[(i0 + 1) % 5] + col[(i1 + 1) % 5]
            ciphertextpairs.append(ciphertextpair)
            applied_rule = True
```

```
        if applied_rule:
            continue

        i0 = 0
        i1 = 0
        j0 = 0
        j1 = 0

        for i in range(5):
            row = key_matrix[i]
            if pair[0] in row:
                i0 = i
                j0 = row.find(pair[0])

            if pair[1] in row:
                i1 = i
                j1 = row.find(pair[1])

        ciphertextpair = key_matrix[i0][j1] + key_matrix[i1][j0]
        ciphertextpairs.append(ciphertextpair)

    return "".join(ciphertextpairs)


def decryption(text, key_matrix):
    plaintextpairs = []
    ciphertextpairs = []
    i = 0
    while i < len(text):
        a = text[i]
        b = text[i + 1]

        ciphertextpairs.append(a + b)
        i += 2

    # rule 2

    for pair in ciphertextpairs:
        applied_rule = False
        for row in key_matrix:
```

```
    if pair[0] in row and pair[1] in row:
        j0 = row.find(pair[0])
        j1 = row.find(pair[1])

        plaintext = row[(j0 + 4) % 5] + row[(j1 + 4) % 5]
        plaintextpairs.append(plaintext)
        applied_rule = True

if applied_rule:
    continue

for j in range(5):
    col = "".join(key_matrix[i][j] for i in range(5))
    if pair[0] in col and pair[1] in col:
        i0 = col.find(pair[0])
        i1 = col.find(pair[1])

        plaintext = col[(i0 + 4) % 5] + col[(i1 + 4) % 5]
        plaintextpairs.append(plaintext)
        applied_rule = True

if applied_rule:
    continue

i0 = 0
i1 = 0
j0 = 0
j1 = 0

for i in range(5):
    row = key_matrix[i]
    if pair[0] in row:
        i0 = i
        j0 = row.find(pair[0])

    if pair[1] in row:
        i1 = i
        j1 = row.find(pair[1])

plaintext = key_matrix[i0][j1] + key_matrix[i1][j0]
plaintextpairs.append(plaintext)
```

```
    return "".join(plaintextpairs)


print('Key :playfire example')
plaintext = 'hide the gold in the tree stump'
plaintext = plaintext.replace(' ', '')
encryption = encryption(plaintext, key_matrix)
print("Decryption :" + encryption)
encrypt = decryption(encryption, key_matrix)
print('Encryption :' + encrypt)
```

Output:

Key :playfire example

Decryption :bmodzbxdnabekudmuixmmouvif

Encryption :hidethegoldinthetrexestump

Source Code for Hill Cipher:

```
import numpy as np
import string

alphabet = string.ascii_lowercase
letter_to_index = dict(zip(alphabet, range(len(alphabet))))
index_to_letter = dict(zip(range(len(alphabet)), alphabet))


def egcd(a, b):
    if a == 0:
        return b, 0, 1
    else:
        gcd, x, y = egcd(b % a, a)
        return gcd, y - (b // a) * x, x


def mat_inv(det, modulus):
    gcd, x, y = egcd(det, modulus)
    if gcd != 1:
        raise Exception('Inverse is not possible')
    else:
        return (x % modulus + modulus) % modulus


def matrix_mod_inv(matrix, modulus):
    det = int(np.round(np.linalg.det(matrix)))
    det_inv = mat_inv(det, modulus)
    matrix_modulus_inv = (
        det_inv * np.round(det * np.linalg.inv(matrix)).astype(int) % modulus
    )
    return matrix_modulus_inv


def encrypt_decrypt(message, key):
    msg = ''
    msg_in_numbers = [letter_to_index[char] for char in message]
    split_p = [
        msg_in_numbers[i:i + len(key)]
        for i in range(0, len(msg_in_numbers), len(key))
```

```
    ]
    for P in split_p:
        P = np.transpose(np.asarray(P))[:, np.newaxis]
        while len(P) != len(key):
            P = np.append(P, letter_to_index[" "])[:, np.newaxis]
        numbers = np.dot(key, P) % len(alphabet)
        n = len(numbers)
        for idx in range(n):
            number = numbers[idx][0]
            msg += index_to_letter[number]
    return msg


message = "help"
key = ([[3, 3], [2, 5]])
Kinv = matrix_mod_inv(key, len(alphabet))
encrypted_message = encrypt_decrypt(message, key)
print("Encryption Message : " +encrypted_message.upper())
decrypted_message = encrypt_decrypt(encrypted_message, Kinv)
print("Encryption Message : "+decrypted_message.upper())
```

Output:

Encryption Message : HIAT

Encryption Message : HELP

Source Code Poly-Alphabetic cipher:

```
def Vigenere(text, s, Flag):
    result = ''
    for i in range(len(text)):
        char = text[i]
        if(Flag):
            result += chr((ord(char) - 97 + ord(s[i]) - 97) % 26 + 97)
        else:
            result += chr((ord(char) - ord(s[i]) + 26) % 26 + 97)
    return result


key = "".join(input('Enter the key : ').lower().split())
plain = ''.join(input('Enter PlainText: ').lower().split())
s = ''
catpillar = 0
for i in range(len(plain)):
    s += key[catpillar % len(key)]
    catpillar += 1

Encrypt = Vigenere(plain, s, True)
print('Encryption Message : '+Encrypt)
Decrypt = Vigenere(Encrypt, s, False)
print('Decryption Message :' + Decrypt)
```

Output:

Enter the key : hello

Enter PlainText: hey this is mahin

Encryption Message : oijevpwtdahlty

Decryption Message :heythisismahin

Source Code Vernam cipher:

```
import random
import string

alphabet = string.ascii_lowercase
mp = dict(zip(alphabet, range(len(alphabet))))
mp2 = dict(zip(range(len(alphabet)), alphabet))


def generateKey(length):
    key = ''
    for i in range(length):
        key += chr(random.randint(65, 90))
    return key


def encrypt(message, key):
    encrypted_message = ''
    encrypted_code = []
    for i in range(len(message)):
        xor = mp[message[i]] ^ mp[key[i]]
        encrypted_code.append(xor)
        encrypted_message += mp2[xor % 26]
    return encrypted_message, encrypted_code


def decrypt(encrypted_code, key):
    decrypted_message = ''
    for i in range(len(encrypted_code)):
        xor = encrypted_code[i] ^ mp[key[i]]
        decrypted_message += mp2[xor % 26]
    return decrypted_message
```

```
message = 'oak'
key = generateKey(len(message)).lower()
# key='son'   #coh
encrypted_message, encrypted_code = encrypt(message, key)
decrypted_message = decrypt(encrypted_code, key)
print("Encryption Message: "+encrypted_message)
print("Decryption Message: "+decrypted_message)
```

Output:

Encryption Message: nsd

Decryption Message: oak

Source Code Brute force attack cipher:

```
import string

alphabet = string.ascii_lowercase
mp = dict(zip(alphabet, range(len(alphabet))))
mp2 = dict(zip(range(len(alphabet)), alphabet))


def bruteforce_encrypt(message):
    for i in range(26):
        encrypted_message = ''
        for j in range(len(message)):
            encrypted_message += mp2[(mp[message[j]] + i) % 26]
        print('%d' % i, encrypted_message)


def bruteforce_decrypt(encrypted_message):
    for i in range(26):
        decrypted_message = ''
        for j in range(len(encrypted_message)):
            decrypted_message += mp2[(mp[encrypted_message[j]] - i + 26) % 26]
        print('%d' % i, decrypted_message)


print('-' * 50)
print('Encryption')
print('-' * 50)
bruteforce_encrypt('hello')
print('-' * 50)
print('Decryption')
print('-' * 50)
bruteforce_decrypt('khoor')
print('-' * 50)
```

Output:

Encryption

---

0 hello

1 ifmmp

2 jgnnq

3 khoor

4 lipps

5 mjqqt

6 nkrru

7 olssv

8 pmttw

9 qnuux

10 rovvy

11 spwwz

12 tqxxa

13 uryyb

14 vszzc

15 wtaad

16 xubbe

17 yvccf

18 zwddg

19 axeeh

20 byffi

21 czggj

22 dahhk

23 ebiil

24 fcjjm

25 gdkkn

Decryption

---

0 khoor

1 jgnnq

2 ifmmp

3 hello

4 gdkkn

5 fcjjm

6 ebiil

7 dahhk

8 czggj

9 byffi

10 axeeh

11 zwddg

12 yvccf

13 xubbe

14 wtaad

15 vszzc

16 uryyb

17 tqxxa

18 spwwz

19 rovvy

20 qnuux

21 pmttw

22 olssv

23 nkrru

24 mjqqt

25 lipps

Source Code RSA algorithm:

```
import random

def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

def mod_inverse(a, m):
    m0, x0, x1 = m, 0, 1
    while a > 1:
        q = a // m
        m, a = a % m, m
        x0, x1 = x1 - q * x0, x0
    return x1 + m0 if x1 < 0 else x1

def is_prime(num):
    if num <= 1:
        return False
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            return False
    return True

def generate_keypair(p, q):
    if not (is_prime(p) and is_prime(q)):
        raise ValueError("Both numbers must be prime.")
    elif p == q:
        raise ValueError("p and q cannot be equal")
    n = p * q
    phi = (p - 1) * (q - 1)
    e = random.randrange(1, phi)
    g = gcd(e, phi)
    while g != 1:
```

```
        e = random.randrange(1, phi)
        g = gcd(e, phi)
    d = mod_inverse(e, phi)
    return ((e, n), (d, n))

def encrypt(public_key, plaintext):
    key, n = public_key
    cipher = [pow(ord(char), key, n) for char in plaintext]
    return cipher

def decrypt(private_key, ciphertext):
    key, n = private_key
    plain = [chr(pow(char, key, n)) for char in ciphertext]
    return ''.join(plain)

if __name__ == '__main__':
    p = int(input("Enter a prime number (p): "))
    q = int(input("Enter another prime number (q): "))
    public_key, private_key = generate_keypair(p, q)
    print("Public Key:", public_key)
    print("Private Key:", private_key)

    message = input("Enter a message to encrypt: ")
    encrypted_msg = encrypt(public_key, message)
    print("Encrypted message:", ''.join(map(lambda x: str(x), encrypted_msg)))

    decrypted_msg = decrypt(private_key, encrypted_msg)
    print("Decrypted message:", decrypted_msg)
```

Output:

Enter a prime number (p): 11

Enter another prime number (q): 13

Public Key: (11, 143)

Private Key: (11, 143)

Enter a message to encrypt: hello I am there

Encrypted message: 104134757567761876141217611610413441 34

Decrypted message: hello I am there