- **Lab Report: RSA Algorithm**

Experiment Title:

Implementation of RSA Encryption and Decryption in Python

Objectives

- To understand the **concept of public-key cryptography**.
- To implement **RSA encryption** using small prime numbers.
- To learn how **modular exponentiation** is used in encryption and decryption.

Theory

The **RSA algorithm** (named after Rivest, Shamir, and Adleman) is one of the most popular methods of **asymmetric key encryption**.
It uses **two different keys**:

- **Public Key (e, n)** → used for **encryption**.
- **Private Key (d, n)** → used for **decryption**.

The security of RSA is based on the **difficulty of factoring large prime numbers**.

Steps of RSA Algorithm

1. Choose two prime numbers p and q.
2. Compute n = p × q.
3. Compute Euler's totient function: φ(n) = (p−1) (q−1).
4. Choose encryption key e such that 1 < e < φ(n) and gcd (e, φ(n)) = 1.
5. Find decryption key d such that (d × e) mod φ(n) = 1.
6. **Encryption:** C = (M^e) mod n.
7. **Decryption:** M = (C^d) mod n.

Algorithm

Encryption:

1. Input the message M and public key (e, n).
2. Compute ciphertext: C = (M^e) mod n.

Decryption:

1. Input ciphertext C and private key (d, n).
2. Compute plaintext: M = (C^d) mod n.

Source Code:

```
# RSA Encryption and Decryption Program
p = 17
q = 19
n = p * q
phi = (p - 1) * (q - 1)
e = 5
d = pow(e, -1, phi)

msg = int(input("Enter a number message: "))

# Encryption
cipher = pow(msg, e, n)

# Decryption
decrypted = pow(cipher, d, n)

print("Public key (e, n):", (e, n))
print("Private key (d, n):", (d, n))
print("Encrypted message:", cipher)
print("Decrypted message:", decrypted)
```

Output:

```
Enter a number message: 8
Public key (e, n): (5, 323)
Private key (d, n): (173, 323)
Encrypted message: 145
Decrypted message: 8
```

RSA হলো একটি পাবলিক-কি ক্রিপ্টোগ্রাফি পদ্ধতি, যেখানে একটি কী দিয়ে বার্তা এনক্রিপ্ট করা হয় এবং অন্য কী দিয়ে ডিক্রিপ্ট করা হয়। এই প্রোগ্রামে ছোট প্রাইম সংখ্যা ব্যবহার করে RSA অ্যালগরিদমের মূল ধারণা সহজভাবে প্রদর্শন করা হয়েছে।

Caesar Cipher

1. Objectives

- To understand the concept of the Caesar Cipher in classical cryptography.
- To implement a program for encrypting and decrypting text using Caesar Cipher.
- To observe how simple shift-based substitution can secure text messages.

---

2. Theory

The **Caesar Cipher** is one of the earliest and simplest forms of encryption, used by **Julius Caesar** to send secret messages.
It is a **monoalphabetic substitution cipher**, where each letter in the plaintext is replaced by a letter a fixed number of positions down the alphabet.

Mathematical Representation:
If each letter is represented by a number (`A=0, B=1, …, Z=25`),
then encryption and decryption can be expressed as:

- **Encryption:**

$$C = (P + K) \bmod 26$$

- **Decryption:**

$$P = (C - K) \bmod 26$$

Where,

- P= ***Plaintext letter*** (in numeric form)
- C= Ciphertext letter (in numeric form)
- K= Key (number of shifts)

The Caesar Cipher provides **basic data confidentiality**, though it is vulnerable to **brute-force attacks** (since only 25 keys exist).

Algorithm

1. Take a text from the user.
2. Choose a shift number.
3. Replace each letter by shifting it forward (for encryption).
4. Shift backward (for decryption).
5. Show the result.

Source Code:

```
t = input("Text: ")

s = int(input("Shift: "))

c = lambda x, y: ''.join(chr((ord(i.upper()) - 65 + y) % 26 + 65) if i.isalpha() else i for i in x)

print("Enc:", c(t, s))

print("Dec:", c(c(t, s), -s))
```

Output:

```
Enter text: zisan is good

Shift: 3

Encrypted: CLVDQ LV JRRG

Decrypted: ZISAN IS GOOD
```

Caesar Cipher একটি সহজ এনক্রিপশন পদ্ধতি। এখানে প্রতিটি অক্ষর নির্দিষ্ট সংখ্যক স্থান সরিয়ে গোপন বার্তা তৈরি করা হয়। প্রোগ্রামটি এনক্রিপশন ও ডিক্রিপশন উভয় কাজই করে।

One-Time Pad Cipher

Objective

- To understand the concept of **perfect encryption** using the One-Time Pad.
- To implement a Python program that encrypts and decrypts a message using a random key of the same length as the message.

Theory

The **One-Time Pad (OTP)** is a symmetric encryption technique where:

- Each letter of the message is combined with a letter from a secret key of the **same length**.
- The encryption and decryption use **modular arithmetic on letters.**
- When the key is used only once and kept secret, this cipher is **unbreakable**.

Formulas:

- Encryption: $C = (P + K) \bmod 26$
- Decryption: $P = (C - K) \bmod 26$

Where,

- P= Plaintext letter number (A=0, B=1, …, Z=25)
- K= Key letter number
- C= Ciphertext letter number

Algorithm

1. Take plaintext input.
2. Take a key of the same length as the message.
3. Convert letters to numbers (A=0, B=1, …, Z=25).
4. For encryption: **Add key letters** (mod 26).
5. For decryption: **Subtract key letters** (mod 26).
6. Convert numbers back to letters to get the message.

Source Code:

```
t = input("Text: ").upper()
k = input("Key: ").upper()
f = lambda s, key, m: ''.join(
    chr((ord(a) - 65 + m * (ord(b) - 65)) % 26 + 65) if a.isalpha() else a
    for a, b in zip(s, key)
)
c = f(t, k, 1)
print("Enc:", c)
print("Dec:", f(c, input("Key: ").upper(), -1))
```

Output:

```
Enter Text to Encrypt: ZISAN
Enter the Key (same length): ASDFG
Encrypted: ZAVFT
Enter the Key to Decrypt: ASDFG
Decrypted: ZISAN
```

One-Time Pad একটি নিখুঁত এনক্রিপশন পদ্ধতি। এখানে বার্তার দৈর্ঘ্যের সমান একটি কী ব্যবহার করা হয় এবং প্রতিটি অক্ষর মডুলার গণিত দিয়ে রূপান্তরিত হয়। সঠিক কী ছাড়া বার্তাটি ডিক্রিপ্ট করা সম্ভব নয়।

Playfair Cipher

Experiment Title

Implementation of Playfair Cipher for Encryption and Decryption

---

Objectives

- To understand the **Playfair cipher** encryption technique.
- To learn how **digraph substitution** works using a 5×5 key matrix.
- To implement encryption and decryption using **Python**.

Theory

The **Playfair Cipher** is a classical encryption technique that encrypts pairs of letters (digraphs) instead of single letters. It uses a **5×5 matrix** of alphabets constructed using a **keyword**, where 'I' and 'J' are treated as the same letter.

Steps in Encryption:

1. Construct a 5×5 key matrix using the given key.
2. Divide plaintext into pairs (digraphs).
    - If both letters are the same, insert an "X" between them.
    - If one letter is left, add "X" at the end.
3. For each pair:
    - If both letters are in the same **row**, replace them with the next letters in the same row.
    - If both are in the same **column**, replace them with the letters **below** them.
    - Otherwise, form a **rectangle** and take letters from the same row but opposite corners.

Decryption is the reverse of encryption, moving in the opposite direction in the matrix.

Algorithm

1. Input the **key** and **plaintext message**.
2. Create the **Playfair 5×5 matrix** (I and J combined).
3. Break the plaintext into **digraphs** (pairs of letters).
4. Apply Playfair cipher rules for encryption.
5. For decryption, reverse the encryption rules.

Source Code:

```
def mat(k):
    k = k.upper().replace("J", "I")
    a = "".join(dict.fromkeys(k + "ABCDEFGHIKLMNOPQRSTUVWXYZ"))
    return [a[i:i+5] for i in range(0, 25, 5)]


def pf(m, k, e=1):
    mt = mat(k)
    r = ""
    m = m.upper().replace("J", "I")
    m = "".join(c for c in m if c.isalpha())
    i = 0
    while i < len(m):
        a = m[i]
        b = m[i+1] if i+1 < len(m) else "X"
        if a == b:
            b = "X"
            i += 1
```

```
        else:
            i += 2
        x1, y1 = [(i, j) for i, row in enumerate(mt) for j, c in enumerate(row) if c == a][0]
        x2, y2 = [(i, j) for i, row in enumerate(mt) for j, c in enumerate(row) if c == b][0]
        if x1 == x2:
            r += mt[x1][(y1 + e) % 5] + mt[x2][(y2 + e) % 5]
        elif y1 == y2:
            r += mt[(x1 + e) % 5][y1] + mt[(x2 + e) % 5][y2]
        else:
            r += mt[x1][y2] + mt[x2][y1]
    return r

k = input("Key: ")
m = input("Msg: ")
e = pf(m, k)
print("Enc:", e)
print("Dec:", pf(e, k, -1))
```

Output:

Key: MONARCY

Msg: ZISAN

Enc: XKXDAW

Dec: ZISANX

Playfair Cipher একটি জোড়া-অক্ষর (digraph) ভিত্তিক এনক্রিপশন পদ্ধতি। এটি ৫×৫ ম্যাট্রিক্স ব্যবহার করে অক্ষরগুলির স্থান পরিবর্তন করে বার্তা গোপন করে। এই পদ্ধতিটি সাধারণ সাবস্টিটিউশন সাইফারের চেয়ে অনেক বেশি নিরাপদ।

Monoalphabetic Cipher

Experiment Title

Implementation of Monoalphabetic Cipher for Encryption and Decryption

---

Objectives

- To understand the **Monoalphabetic Cipher** encryption technique.
- To implement a program that **replaces each letter** of the plaintext with another fixed letter.
- To learn how **substitution-based encryption** works in cryptography.

Theory

The **Monoalphabetic Cipher** is a type of **substitution cipher** where each letter in the plaintext is replaced with another letter from a fixed permutation of the alphabet.
The key is a **one-to-one mapping** between the plaintext alphabet and the ciphertext alphabet.

For example:
If the mapping is:

```
Plain:  ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher: QWERTYUIOPASDFGHJKLZXCVBNM
```

Then,
Plaintext: ZISAN
Ciphertext: MOLQF

Since every letter always maps to the same symbol, it's easy to implement but can be broken through frequency analysis.

Algorithm

1. Define the **plaintext alphabet (a–z)**.
2. Define a **cipher alphabet** (a shuffled version of a–z).
3. Replace each plaintext letter using the cipher alphabet.
4. For decryption, reverse the substitution using the original alphabet.

Source Code:

```
a = "abcdefghijklmnopqrstuvwxyz"

b = "qwertyuiopasdfghjklzxcvbnm"

m = input("Message: ")

# Encryption

e = "".join([b[a.index(c)] if c in a else c for c in m.lower()])

# Decryption

d = "".join([a[b.index(c)] if c in b else c for c in e])

print("Enc:", e, "\nDec:", d)
```

Output:

```
Message: Zisan
Encrypted: molqf
Decrypted: zisan
```

Monoalphabetic Cipher একটি সাধারণ প্রতিস্থাপন (substitution) এনক্রিপশন পদ্ধতি যেখানে প্রতিটি অক্ষর একটি নির্দিষ্ট নতুন অক্ষর দিয়ে প্রতিস্থাপিত হয়। এটি সহজে বাস্তবায়নযোগ্য হলেও খুব নিরাপদ নয়, কারণ অক্ষর-বার্তার পুনরাবৃত্তি দেখে এটি ভাঙা সম্ভব।

Hill Cipher

Experiment Title

Implementation of Hill Cipher for Encryption and Decryption

Objectives

- To understand the concept of **matrix-based encryption** in cryptography.
- To implement the **Hill Cipher** using Python.
- To apply **linear algebra (matrix multiplication modulo 26)** in message encryption.

Theory

The **Hill Cipher** is a classical **polygraphic substitution cipher** based on **linear algebra**.
It encrypts a block of letters (e.g., pairs or triplets) using **matrix multiplication** and modular arithmetic.

Key Concept:

- Each letter is represented as a number (A=0, B=1, …, Z=25).
- Plaintext letters are grouped into vectors.
- Each vector is multiplied by the key matrix.
- Results are taken modulo 26 to get the ciphertext vector.

Decryption uses the **inverse of the key matrix** (mod 26).

Mathematical Formulas

- **Encryption:**

$$C = (K \times P) \bmod 26$$

- **Decryption:**

$$P = (K^{-1} \times C) \bmod 26$$

Where:

- K= Key matrix
- P= Plaintext vector
- C= Ciphertext vector

Algorithm

1. Represent plaintext letters as numbers (A=0 to Z=25).
2. Divide plaintext into blocks equal to the matrix order.
3. Multiply each block by the key matrix.
4. Take modulo 26 of the result.
5. Convert back to letters to get ciphertext.
6. For decryption, use the inverse key matrix (mod 26).

Source Code:

```
import numpy as np

k = np.array([[3, 3], [2, 5]])

def hi(m, enc=True):
    m = m.upper().replace(" ", "") + ("X" * (len(m) % 2))
    K = k if enc else (pow(int(round(np.linalg.det(k))), -1, 26) *
                np.array([[k[1,1], -k[0,1]], [ -k[1,0], k[0,0]]])) % 26
    r = ""
    for i in range(0, len(m), 2):
        r += "".join(chr(int(x) + 65)
                for x in np.dot(K, [ord(m[i]) - 65, ord(m[i+1]) - 65]) % 26)
    return r

x = input("Msg: ")
e = hi(x)
print("Enc:", e)
print("Dec:", hi(e, False))
```

Output:

Message: is it bad or not

Encrypted: ACDHDCZYMVVT

Decrypted: ISITBADORNOT

Hill Cipher একটি ম্যাট্রিক্স-ভিত্তিক এনক্রিপশন পদ্ধতি যেখানে বার্তার অক্ষরগুলো সংখ্যা আকারে রূপান্তর করে ম্যাট্রিক্স গুণের মাধ্যমে এনক্রিপ্ট করা হয়। এটি সাধারণ সাবস্টিটিউশন সাইফারের চেয়ে অনেক বেশি নিরাপদ।

Brute-Force Attack on Caesar Cipher

Experiment Title

Implementation of Brute-Force Attack to Recover Plaintext from a Caesar Cipher

---

Objectives

- To understand how a **brute-force attack** can break a Caesar cipher.
- To implement a simple Python program that **tries all possible shifts** and highlights likely plaintexts.
- To practice detecting meaningful results by checking for common English words.

Theory

A **Caesar cipher** shifts every letter by a fixed number (0–25). Because there are only 26 possible shifts, an attacker can **try every shift** (brute force) to recover the plaintext. To help identify the correct result, the program checks for the presence of common English words (like “the”, “and”, “is”, etc.) in each trial plaintext.

Algorithm

1. Read the ciphertext from the user.
2. For each possible shift `s` from 0 to 25:
    - Shift each letter **back** by `s` (i.e., try decrypting with key `s`).
    - Build the candidate plaintext.
    - Check if any common words appear in the candidate (to mark likely correct results).
    - Print the candidate, optionally highlighting likely ones.

Because the search space is only 26 shifts, brute force is trivial and fast for Caesar.

Source code:

```
ct=input("Cipher Text:").lower()

W={'the','and','is','you','meet','attack','secret','hello','password','flag'}

for s in range(26):
pt=''.join(chr((ord(c)-97-s)%26+97) if 'a'<=c<='z' else c for c in ct)

print(f"{s:2}: " + (f"\x1b[4m{pt}\x1b[0m" if any(w in pt.split() for w in W) else pt))
```

Output:

```
Cipher Text:clvdq lv jrrg
0: clvdq lv jrrg
1: bkucp ku iqqf
2: ajtbo jt hppe
3: zisan is good
4: yhrzm hr fnnc
5: xgqyl gq emmb
```

6: wfpxk fp dlla

7: veowj eo ckkz

8: udnvi dn bjjy

9: tcmuh cm aiix

10: sbltg bl zhhw

11: raksf ak yggv

12: qzjre zj xffu

13: pyiqd yi weet

14: oxhpc xh vdds

15: nwgob wg uccr

16: mvfna vf tbbq

17: luemz ue saap

18: ktdly td rzzo

19: jsckx sc qyyn

20: irbjw rb pxxm

21: hqaiv qa owwl

22: gpzhu pz nvvk

23: foygt oy muuj

24: enxfs nx ltti

25: dmwer mw kssh

GCD Using Euclidean Algorithm

Experiment Title:

Implementation of **Euclidean Algorithm** to find the Greatest Common Divisor (GCD)

Objective:

- To understand how the **Euclidean Algorithm** works to find the GCD of two numbers.
- To write a **Python program** that calculates the GCD using repeated division.

Theory:

The **Euclidean Algorithm** is a simple and efficient method to find the **GCD (Greatest Common Divisor)** of two numbers.
It is based on the rule:
GCD(a, b) = GCD(b, a mod b),
until **b = 0**, then **GCD = a**.

This means we repeatedly replace the larger number by the remainder of dividing it by the smaller number until the remainder becomes zero.

Algorithm:

1. Start with two numbers `a` and `b`.
2. While `b` is not zero:
    - Replace `a` with `b`.
    - Replace `b` with `a % b`.
3. When `b` becomes zero, `a` is the GCD.
4. Display the result.

Python Code:

```
# Function to find GCD using Euclidean Algorithm

def gcd(a, b):

    while b != 0:

        a, b = b, a % b

    return a

# Take input from user

num1 = int(input("Enter first number: "))

num2 = int(input("Enter second number: "))

# Display result

print("GCD of", num1, "and", num2, "is:", gcd(num1, num2))
```

Output:

```
Enter first number: 25

Enter second number: 250

GCD of 25 and 250 is: 25
```

Extended Euclidean Algorithm

Experiment Title:

Implementation of **Extended Euclidean Algorithm** to find GCD and coefficients

---

Objective:

- To understand how the **Extended Euclidean Algorithm (EEA)** works.
- To implement a Python program that finds the **GCD** of two numbers and integers **x** and **y** such that:

$$a \cdot x + b \cdot y = \text{GCD}(a, b)$$

- To prepare for applications in **modular inverses** used in cryptography.

Theory:

The **Extended Euclidean Algorithm** extends the standard Euclidean Algorithm by also finding integers **x** and **y** such that:

$$a \cdot x + b \cdot y = \text{GCD}(a, b)$$

Steps:

1. If $b = 0$, then $\text{GCD}(a, b) = a$and coefficients are $x = 1, y = 0$.
2. Otherwise, recursively compute:

$$g, x_1, y_1 = \text{egcd}(b, a \bmod b)$$

3. Update coefficients:

$$x = y_1, y = x_1 - \lfloor \frac{a}{b} \rfloor \cdot y_1$$

4. Return g, x, y.

This algorithm is widely used in **cryptography**, especially to find **modular inverses** in RSA and other public-key systems.

Algorithm:

1. Input two numbers `a` and `b`.
2. Check if `b = 0`: return `gcd = a, x = 1, y = 0`.
3. Otherwise, recursively call `egcd(b, a % b)`.
4. Update coefficients and return `gcd, x, y`.
5. Display results.

Python Code:

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

# Input from user
a, b = map(int, input("Enter a b: ").split())

# Compute gcd and coefficients
g, x, y = egcd(a, b)
print(f"gcd={g}, x={x}, y={y}")
```

Output:

```
Enter a b: 20 23
gcd=1, x=-8, y=7
```

DES (XOR) Encryption

Experiment Title

Implementation of **DES-style XOR Encryption and Decryption** in Python

---

Objectives

- To understand **symmetric key encryption**.
- To implement a simple **DES-style XOR encryption** in Python.
- To learn the process of **encrypting and decrypting messages** using a secret key.

Theory

DES (Data Encryption Standard) is a **symmetric-key encryption** technique.

- In symmetric encryption, the **same key** is used for both encryption and decryption.
- This simplified version uses **XOR operation** between each character of the plaintext and the key.
- Encryption:

$$C_i = P_i \oplus K_i$$

- Decryption:

$$P_i = C_i \oplus K_i$$

Where $\oplus$denotes the XOR operation.

Algorithm

1. Take a **key** from the user.
2. Take the **plaintext message** from the user.
3. Encrypt each character:
    - Convert characters to ASCII.
    - XOR with the corresponding character of the key (cycling the key if shorter than message).
4. Convert the encrypted bytes to **hex** for readability.
5. Decrypt by XORing the encrypted bytes with the same key.
6. Display both **encrypted** and **decrypted** results.

Python Code

```
# Input key and plaintext

k = input("Key: ")

t = input("Text: ")

# Encryption using XOR

e = bytes([ord(t[i]) ^ ord(k[i % len(k)]) for i in range(len(t))])

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


# Decryption using XOR

d = bytes([e[i] ^ ord(k[i % len(k)]) for i in range(len(e))])

print("Decrypted:", d.decode())
```

Output:

Key: Zisan

Text: I he good

Encrypted: 13491b044e3d061c05

Decrypted: I he good

এ প্রোগ্রামটি **সিমেট্রিক কী এনক্রিপশন** দেখায়, যেখানে একই কী দিয়ে বার্তা এনক্রিপ্ট এবং ডিক্রিপ্ট করা হয়। এখানে XOR অপারেশন ব্যবহার করে বার্তা নিরাপদ রাখা হয়েছে।

Diffie-Hellman Key Exchange with Simple XOR Encryption

Experiment Title

Implementation of **Diffie-Hellman Key Exchange** for secure message encryption and decryption

---

Objectives

- To understand the **Diffie-Hellman key exchange** protocol.
- To implement a shared secret generation between two users (Alice and Bob).
- To encrypt and decrypt messages using the shared secret as a **symmetric key**.

Theory

The **Diffie-Hellman (DH) Key Exchange** allows two parties to establish a **shared secret key** over an insecure channel.

- **Public parameters:** p(prime modulus) and g(generator).
- **Private keys:** Randomly chosen by each user (Alice = a, Bob = b).
- **Public keys:** Computed as:

$$A = g^a \bmod p, B = g^b \bmod p$$

- **Shared secret:** Each user computes:

$$s = B^a \bmod p = A^b \bmod p$$

Once the shared secret is computed, it can be used as a **symmetric key** for encrypting and decrypting messages.

In this example, a **simple XOR** is used for demonstration:

$$\text{Encrypted character} = \text{Message character} \oplus \text{Shared secret}$$

Algorithm

1. Agree on **public parameters** $p$and g.
2. Each user selects a **private key**.
3. Compute the **public key** for each user.
4. Exchange public keys and compute the **shared secret**.
5. Encrypt the message using XOR with the shared secret.
6. Decrypt by XORing again with the shared secret.

Python Code

```
# Diffie-Hellman Key Exchange

p, g = 23, 5  # Public parameters



# Key generation
```

```
a = int(input("Alice's private key: "))  # Alice's private key
b = int(input("Bob's private key: "))    # Bob's private key

A = (g ** a) % p  # Alice's public key
B = (g ** b) % p  # Bob's public key

# Shared secret
s1 = (B ** a) % p  # Alice computes
s2 = (A ** b) % p  # Bob computes

print(f"Shared secret: {s1}")
# Simple XOR encryption
message = input("Message: ")
key = s1  # Use shared secret as key
# Encrypt
encrypted = ''.join(chr(ord(c) ^ key) for c in message)
print(f"Encrypted: {encrypted}")
# Decrypt
decrypted = ''.join(chr(ord(c) ^ key) for c in encrypted)
print(f"Decrypted: {decrypted}")
```

Output:

Alice's private key: 6

Bob's private key: 15

Shared secret: 2

Message: I am Zisan

Encrypted: K"co"Xkqcl

Decrypted: I am Zisan

Diffie-Hellman key exchange ব্যবহার করে দুই পক্ষ সহজেই একটি **shared secret key** তৈরি করতে পারে। এরপর এই key ব্যবহার করে বার্তা এনক্রিপ্ট এবং ডিক্রিপ্ট করা যায়। এখানে XOR অপারেশন দিয়ে ধারণা দেখানো হয়েছে।

Modular Arithmetic – Checking Primitive Roots

Experiment Title

Implementation of **Modular Arithmetic** to check for a **primitive root** modulo n

---

Objectives

- To understand the concept of **modular arithmetic**.
- To determine whether a given number is a **primitive root modulo n**.
- To apply **powers and modulo operations** in encryption-related computations.

Theory

A number $g$is called a **primitive root modulo n** if the powers of $g$generate all numbers from 1to $n-1$modulo n.

Formally:

$$\{g^1 mod n, g^2 mod n, \dots, g^{n-1} mod n\} = \{1,2, \dots, n-1\}$$

Primitive roots are important in cryptography, especially for **Diffie-Hellman key exchange** and other modular exponentiation-based algorithms.

Algorithm

1. Take inputs $n$and $g$from the user.
2. For each $i$from 1 to $n-1$:
   - Compute $r = g^i \text{mod } n$.
   - Print the result.
   - If $r = 1$before reaching $i = n-1$, then $g$is **not a primitive root**.
3. If the loop completes without early repetition, g**is a primitive root of n**.

Python Code

```
# Input n and g
n, g = map(int, input("Enter n and g: ").split())

print(f"\nChecking if {g} is primitive root mod {n}:")

for i in range(1, n):
    r = pow(g, i, n)
    print(f"{g}^{i} mod {n} = {r}")
    if r == 1 and i != n-1:
        print(f"  Not primitive root (repeats at {i})")
        break
```

```
else:
    print(f"   {g} is primitive root of {n}")
```

Output:

```
Enter n and g: 7 3

Checking if 3 is primitive root mod 7:
3^1 mod 7 = 3
3^2 mod 7 = 2
3^3 mod 7 = 6
3^4 mod 7 = 4
3^5 mod 7 = 5
3^6 mod 7 = 1
3 is primitive root of 7
```

এই প্রোগ্রামটি একটি সংখ্যা *g* কি **primitive root modulo n** তা পরীক্ষা করে। Primitive root হলো এমন একটি সংখ্যা যার ঘাত mod n সমস্ত সংখ্যা 1 থেকে n-1 পর্যন্ত তৈরি করে। এটি ক্রিপ্টোগ্রাফিতে খুব গুরুত্বপূর্ণ।

ElGamal Encryption and Decryption

Experiment Title

Implementation of **ElGamal Public-Key Cryptography** in Python

---

Objectives

- To understand the concept of **public-key cryptography**.
- To implement the **ElGamal encryption and decryption algorithm**.
- To learn **modular exponentiation** and **modular inverse** for encryption purposes.

Theory

ElGamal is an **asymmetric encryption** algorithm based on **modular arithmetic** and the **discrete logarithm problem**.

Key components:

- **Public parameters:** prime p, generator g, and public key $y = g^x \bmod p$.
- **Private key:** x(kept secret).
- **Encryption:** For a message $m$and random k:

$$c_1 = g^k \bmod p, c_2 = m \cdot y^k \bmod p$$

- **Decryption:** Using private key x:

$$m = c_2 \cdot (c_1^x)^{-1} \bmod p$$

Where $(c_1^x)^{-1}$ is the **modular inverse** of c_1^x modulo p.

Algorithm

1. Input public parameters p, g, x, y and plaintext m.
2. Choose a random k for encryption.
3. Compute:
   - $c_1 = g^k \bmod p$
   - $c_2 = m \cdot y^k \bmod p$
4. Send ciphertext (c_1, c_2).
5. Decrypt using:
   - $s = c_1^x \bmod p$
   - $m = c_2 \cdot s^{-1} \bmod p$
6. Display encrypted and decrypted messages.

Python Code

```
# ElGamal Encryption/Decryption

p, g, x, y = map(int, input("Enter p,g,x,y: ").split())  # x = private, y = public

m = int(input("Message: "))


# Encryption

k = 7  # can also be random

c1 = pow(g, k, p)

c2 = (m * pow(y, k, p)) % p

print(f"Encrypted: ({c1}, {c2})")


# Decryption

s = pow(c1, x, p)

m_dec = (c2 * pow(s, p - 2, p)) % p  # Modular inverse using Fermat's little theorem

print(f"Decrypted: {m_dec}")
```

Output:

```
Enter p,g,x,y: 23 5 6 8

Message: 10
```

Encrypted: (17, 5)

Decrypted: 10

ElGamal ক্রিপ্টোসিস্টেম ব্যবহার করে বার্তা **এনক্রিপ্ট এবং ডিক্রিপ্ট** করা যায়। এখানে **প্রাইভেট কী** ব্যবহার করে **শেয়ার করা পাবলিক কী** থেকে বার্তা পুনরুদ্ধার করা হয়।

Rail Fence Transposition Cipher

Experiment Title

Implementation of **Rail Fence Cipher** for Encryption and Decryption

---

Objectives

- To understand the **transposition cipher** technique using rails.
- To implement the **Rail Fence Cipher** in Python for encrypting a message.
- To decrypt the ciphertext back to plaintext using the same rail pattern.

Theory

The **Rail Fence Cipher** is a type of **transposition cipher**:

- It rearranges the letters of plaintext into a **zigzag pattern** on multiple rails (rows).
- The message is then read row by row to get the ciphertext.

For example, with **2 rails**:

```
Plaintext: HELLO WORLD
Remove spaces: HELLOWORLD

Rail 1: H L O O L
Rail 2: E L W R D
Ciphertext: HLOOLELWRD
```

Decryption involves reconstructing the rails and reading them in **zigzag order** to recover the original message.

Algorithm

1. Remove spaces from plaintext.
2. For **2 rails**:
    - Rail 1: take characters at even indices.
    - Rail 2: take characters at odd indices.
3. Combine rails to form **ciphertext**.
4. For decryption:
    - Split ciphertext according to rail lengths.
    - Reconstruct original positions to get plaintext.
5. Display encrypted and decrypted messages.

Python Code

```
# Input text

text = input("Enter text: ").replace(" ", "")

depth = 2  # Number of rails



# Encryption

rail1 = text[0::2]

rail2 = text[1::2]

encrypted = rail1 + rail2



# Decryption

half = (len(encrypted) + 1) // 2

decrypted = [''] * len(encrypted)

decrypted[0::2] = encrypted[:half]

decrypted[1::2] = encrypted[half:]



# Display results

print(f"Encrypted: {encrypted}")

print(f"Decrypted: {''.join(decrypted)}")
```

Output:

Enter text: hello world

Encrypted: hloolelwrd

Decrypted: helloworld

Rail Fence Cipher একটি সহজ ট্রান্সপোজিশন সাইফার। বার্তাকে রেল আকারে সাজিয়ে এনক্রিপ্ট করা হয়। পুনরায় একই রেল আকার ব্যবহার করে বার্তা ডিক্রিপ্ট করা যায়।

Transposition Cipher

Experiment Title:

Implementation of Transposition Cipher for Encryption and Decryption

---

Objectives

- To understand the **transposition technique** in cryptography.
- To implement a **columnar transposition cipher** for encrypting a message.
- To strengthen knowledge of **permutation-based encryption** rather than substitution-based.

Theory

The **Transposition Cipher** is a classical encryption method in which the **positions of letters** in a message are **rearranged according to a fixed key**.
Unlike substitution ciphers, it **does not replace letters** — instead, it **changes their order** to make the message unreadable.

For example:
Plaintext: HELLO
Key: 3 (number of columns)

Arrange in rows of 3:

H	E	L
L	O	X

(Read column-wise → **H L E O L X**)
Ciphertext: HLEOLX

Algorithm

Encryption Steps

1. Choose a key (either a word or number defining column order).
2. Write the plaintext row by row into a matrix with the chosen number of columns.
3. Fill remaining cells with padding (e.g., ‘X’).
4. Rearrange the columns according to the alphabetical order of the key.
5. Read the message column by column to form the ciphertext.

Decryption Steps

1. Determine the number of rows and columns based on the key.
2. Write the ciphertext column by column into the matrix.
3. Rearrange the columns back to their original order.
4. Read the matrix row by row to get the plaintext.

Source Code:

```
def encrypt(msg, key):
    msg = msg.replace(" ", "").upper()
cols = len(key)
msg += "X" * ((cols - len(msg) % cols) % cols)
matrix = [msg[i:i+cols] for i in range(0, len(msg), cols)]
order = sorted(range(cols), key=lambda i: key[i])
return ''.join(matrix[r][c] for c in order for r in range(len(matrix)))
m = input("Message: ")
k = input("Key: ")
print("Encrypted:", encrypt(m, k))
```

Output:

Message: i am back

Key: zim

Encrypted: AAXMCXIBK

Hill Cipher

Experiment Title:

Implementation of Hill Cipher for Encryption and Decryption