INDEX

PRACTICAL-1

AIM: Write a program to implement (i) Chinese remainder theorem.

THEORY: The Chinese Remainder Theorem (CRT) is a mathematical theorem that provides a way to solve a system of linear congruences with pairwise coprime moduli. The theorem states that if you have a system of congruences:

$x = a_1(\text{mod } m_1)$,

$x = a_2(\text{mod } m_2)$,

…….,

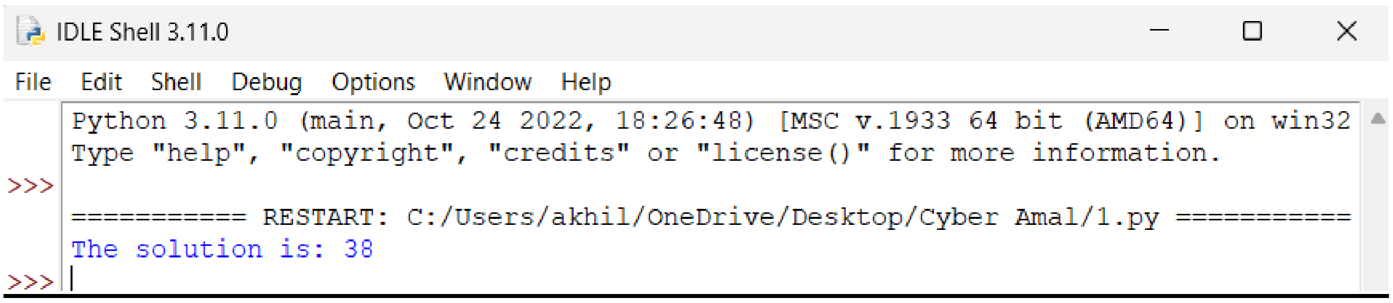
$x = a_n(\text{mod } m_n)$,

where $m_1,m_2,\ldots\ldots,m_n$ are pairwise coprime (i.e., $\gcd(m_i,m_j) = 1$ for $i \neq 1$ then there exists a unique solution $M = m_1 \cdot m_2 \cdots m_k$

CODE:

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


def mod_inverse(a, m):
    g, x, y = extended_gcd(a, m)
    if g != 1:
        raise Exception("The modular inverse does not exist")
    else:
        return x % m


def chinese_remainder_theorem(congruences):
    # Compute M, the product of all moduli
    M = 1
    for _, m in congruences:
        M *= m

    result = 0
    for a, m in congruences:
        # Compute Mi and xi
        Mi = M // m
        xi = mod_inverse(Mi, m)

        # Calculate the contribution of each congruence
        result += a * Mi * xi

    return result % M

# Example usage:
congruences = [(2, 3), (3, 5), (2, 7)]
result = chinese_remainder_theorem(congruences)
print("The solution is:", result)
```

OUTPUT:



```
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=========== RESTART: C:/Users/akhil/OneDrive/Desktop/Cyber Amal/1.py ===========
The solution is: 38
>>> |
```

Conclusion:

In conclusion, the Chinese Remainder Theorem (CRT) is a powerful mathematical tool for solving systems of linear congruences when the moduli involved are pairwise coprime. It provides a method to efficiently find a unique solution modulo the product of the moduli. The CRT has applications in various fields, including number theory, cryptography, and computer science, where it is used to optimize computations involving modular arithmetic and solve modular equations.

II <u>AIM:</u> write a program to implement (ii) Fermat’s little theorem.

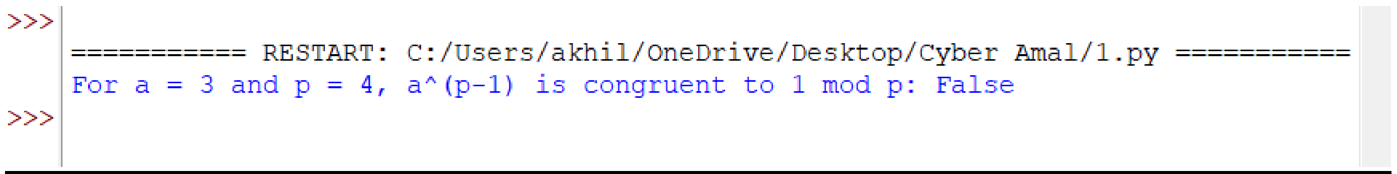
THEORY:

Fermat's Little Theorem is a fundamental result in number theory, named after the French mathematician Pierre de Fermat. The theorem states that if *p* is a prime number and *a* is an integer not divisible by *p*, then $a^{p-1} \equiv 1 (\text{mod}) ap-1 \equiv 1 (\text{mod} p)$.

CODE:

```
def power_mod(base, exponent, modulus):
    result = 1
    base = base % modulus

    while exponent > 0:
        if exponent % 2 == 1:
            result = (result * base) % modulus
        exponent //= 2
        base = (base * base) % modulus

    return result

def fermat_little_theorem(a, p):
    if not (1 < a < p):
        raise ValueError("a must be an integer between 1 and p-1")

    result = power_mod(a, p - 1, p)
    return result == 1

# Example usage:
p = 7
a = 3
result = fermat_little_theorem(a, p)
print(f"For a = {a} and p = {p}, a^(p-1) is congruent to 1 mod p:", result)
```

OUTPUT:



```
>>>
    =========== RESTART: C:/Users/akhil/OneDrive/Desktop/Cyber Amal/1.py ===========
    For a = 3 and p = 4, a^(p-1) is congruent to 1 mod p: False
>>>
```

conclusion: In conclusion, Fermat's Little Theorem states that if (p) is a prime number and (a) is an integer not divisible by (p), then (a^{p-1}= mod p). This theorem is a fundamental result in number theory and has important applications in modular arithmetic, primality testing, and cryptography. It provides a concise and efficient way to verify the primality of a number and is widely utilized in various mathematical and computational fields.

PRACTICAL-2

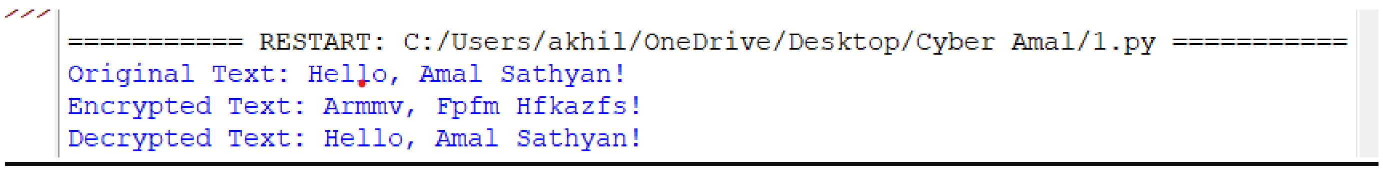
AIM: Write a program to implement the (i) Affine cipher for encryption and decryption.

THEORY: The Affine Cipher is a type of monoalphabetic substitution cipher, where each letter in an alphabet is mapped to its numeric equivalent, encrypted using a simple mathematical function, and converted back to a letter. The encryption function for the Affine Cipher is $E(x)=(ax+b) \bmod m$, where a and b are the key parameters, and m is the size of the alphabet. For decryption $D(x)=a^{-1}\cdot(x-b) \bmod m$

CODE:

```
def gcd(a, b):
    while b:
        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 encrypt(plain_text, a, b, m):
    cipher_text = ''
    for char in plain_text:
        if char.isalpha():
            is_upper = char.isupper()
            char = char.lower()
            encrypted_char = chr(((ord(char) - ord('a')) * a + b) % 26 + ord('a'))
            cipher_text += encrypted_char.upper() if is_upper else encrypted_char
        else:
            cipher_text += char
    return cipher_text


def decrypt(cipher_text, a, b, m):
    a_inv = mod_inverse(a, m)
    plain_text = ''
    for char in cipher_text:
        if char.isalpha():
            is_upper = char.isupper()
            char = char.lower()
            decrypted_char = chr((a_inv * (ord(char) - ord('a') - b)) % 26 + ord('a'))
            plain_text += decrypted_char.upper() if is_upper else decrypted_char
        else:
            plain_text += char
    return plain_text

# Example usage:
plain_text = "Hello, Amal Sathyan!"
a, b, m = 3, 5, 26
cipher_text = encrypt(plain_text, a, b, m)
decrypted_text = decrypt(cipher_text, a, b, m)

print("Original Text:", plain_text)
print("Encrypted Text:", cipher_text)
print("Decrypted Text:", decrypted_text)
```

OUTPUT:



```
=========== RESTART: C:/Users/akhil/OneDrive/Desktop/Cyber Amal/1.py ===========
Original Text: Hello, Amal Sathyan!
Encrypted Text: Armmv, Fpfm Hfkazfs!
Decrypted Text: Hello, Amal Sathyan!
```

Conclusion:

In conclusion, the Affine Cipher is a straightforward encryption technique that combines the simplicity of a substitution cipher with a mathematical transformation. While it is relatively easy to implement and understand, its security is limited due to the small key space, making it vulnerable to brute force attacks. Nonetheless, the Affine Cipher remains a valuable educational tool for introducing basic principles of encryption.

II. AIM: Write a program to implement (ii) Rail fence cipher the for encryption and decryption.

THEORY:

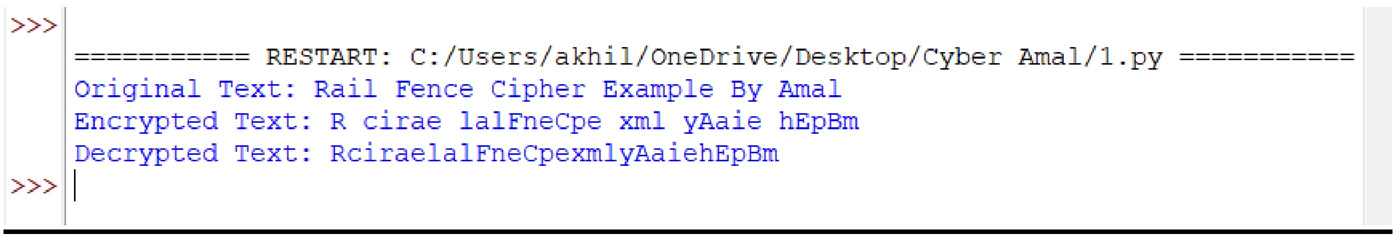
The Rail Fence Cipher is a transposition cipher that works by writing the plaintext in a zigzag pattern across a set number of rows, then reading it back in a different order. The key parameter is the number of rows used in the zigzag pattern.

- **Encryption:** The plaintext is written in a zigzag pattern across the specified number of rows, and then the ciphertext is obtained by reading the characters row by row.
- **Decryption:** An empty rail fence is created with the same number of rows, and the ciphertext is placed in the zigzag pattern. The original plaintext is then obtained by reading the characters in the original order.

CODE:

```
def encrypt_rail_fence(plain_text, num_rows):
    rail = [''] * num_rows
    direction = 1  # 1 for down, -1 for up
    row = 0

    for char in plain_text:
        rail[row] += char
        row += direction

        if row == num_rows - 1 or row == 0:
            direction *= -1

    cipher_text = ''.join(rail)
    return cipher_text


def decrypt_rail_fence(cipher_text, num_rows):
    rail = [''] * num_rows
    direction = 1
    row = 0

    for char in cipher_text:
        rail[row] += ' '
        row += direction
        if row == num_rows - 1 or row == 0:
            direction *= -1

    idx = 0
    for i in range(num_rows):
        for j in range(len(rail[i])):
            rail[i] = rail[i][:j] + cipher_text[idx] + rail[i][j + 1:]
            idx += 1

    plain_text = ''.join(rail)
    return plain_text.replace(' ', '')

# Example usage:
plain_text = "Rail Fence Cipher Example by Amal"
num_rows = 3

cipher_text = encrypt_rail_fence(plain_text, num_rows)
decrypted_text = decrypt_rail_fence(cipher_text, num_rows)

print("Original Text:", plain_text)
print("Encrypted Text:", cipher_text)
print("Decrypted Text:", decrypted_text)
```

<u>OUTPUT:</u>



```
>>>
    =========== RESTART: C:/Users/akhil/OneDrive/Desktop/Cyber Amal/1.py ===========
    Original Text: Rail Fence Cipher Example By Amal
    Encrypted Text: R cirae lalFneCpe xml yAaie hEpBm
    Decrypted Text: RciraelalFneCpexmlyAaiehEpBm
>>> |
```

<u>Conclusion:</u>

The Rail Fence Cipher is a simple transposition cipher that provides a basic level of security. However, it is vulnerable to frequency analysis and other attacks due to its regular patterns. While it is not suitable for serious encryption needs, it serves as an interesting educational example of transposition ciphers and their implementation.

III.AIM: Write a program to implement (iii) Simple columnar technique the for encryption and decryption.

THEORY:

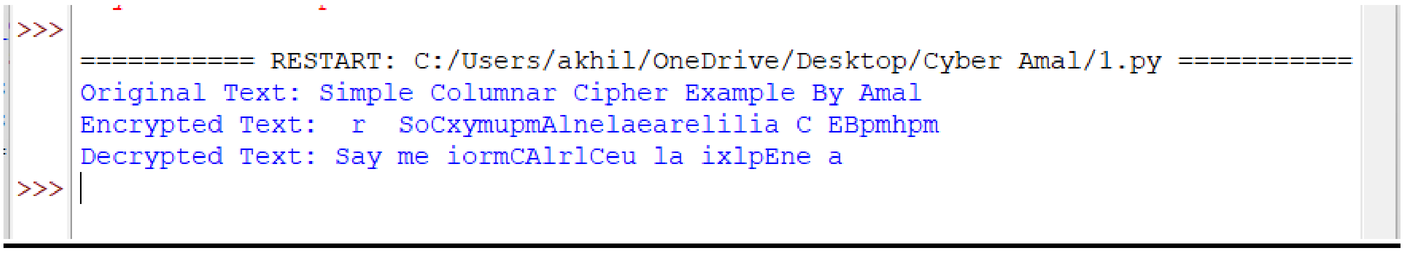
The Simple Columnar Transposition Cipher is a classical transposition cipher that rearranges the characters of a message by writing them out in rows of a fixed length and then reading them out again in columns, but in a different order. The key is the permutation order of the columns.

- **Encryption:** The plaintext is arranged in a matrix column by column based on the key permutation. The ciphertext is then obtained by reading out the matrix column by column.
- **Decryption:** An empty matrix is created with the same dimensions as the original matrix. The ciphertext is filled into the matrix based on the key permutation, and the original plaintext is obtained by reading out the matrix row by row.

CODE:

```
def encrypt_columnar(plain_text, key):
    order = sorted(range(len(key)), key=lambda k: key[k])
    cipher_text = ''

    for col in order:
        for row in range(len(plain_text) // len(key) + 1):
            idx = col + row * len(key)
            if idx < len(plain_text):
                cipher_text += plain_text[idx]

    return cipher_text

def decrypt_columnar(cipher_text, key):
    order = sorted(range(len(key)), key=lambda k: key[k])
    num_cols = len(key)
    num_rows = len(cipher_text) // num_cols
    matrix = [[''] * num_cols for _ in range(num_rows)]

    idx = 0
    for col in order:
        for row in range(num_rows):
            matrix[row][col] = cipher_text[idx]
            idx += 1

    plain_text = ''.join([''.join(row) for row in matrix])
    return plain_text

# Example usage:
plain_text = "Simple Columnar Cipher Example By Amal"
key = "COLUMNAR"

cipher_text = encrypt_columnar(plain_text, key)
decrypted_text = decrypt_columnar(cipher_text, key)

print("Original Text:", plain_text)
print("Encrypted Text:", cipher_text)
print("Decrypted Text:", decrypted_text)
```

OUTPUT:



```
>>>
=========== RESTART: C:/Users/akhil/OneDrive/Desktop/Cyber Amal/1.py ===========
Original Text: Simple Columnar Cipher Example By Amal
Encrypted Text:  r  SoCxymupmAlnelaearelilia C EBpmhpm
Decrypted Text: Say me iormCAlrlCeu la ixlpEne a
>>>
```

Conclusion:

The Simple Columnar Transposition Cipher provides a basic level of security by rearranging the order of characters in the plaintext. However, it is susceptible to frequency analysis and other attacks. While it may not be suitable for high-security applications, it serves as an educational example of a transposition cipher.

IV.AIM: Write a program to implement (iv) vermin cipher for encryption and decryption.

THEORY:

The Vernam Cipher, also known as the one-time pad, is a symmetric-key encryption algorithm that provides perfect secrecy when used correctly. The key used in the Vernam Cipher must be as long as the message and generated using a true random process. Each character in the plaintext is combined (XORed) with the corresponding character in the key to produce the ciphertext.

The XOR (exclusive or) operation is used because it is reversible. When the same key is XORed with the ciphertext, the original plaintext is obtained. This property, along with the requirement for a truly random key, makes the Vernam Cipher theoretically unbreakable.

The key points of the Vernam Cipher are:

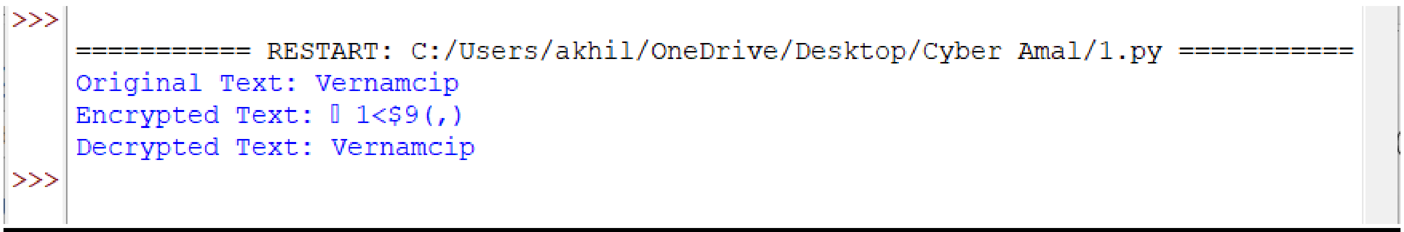
1. **Key Length:** The key must be at least as long as the plaintext and should never be reused for different messages.
2. **Perfect Secrecy:** If the key is truly random, and each key is used only once, the Vernam Cipher achieves perfect secrecy, making it information-theoretically secure.

CODE:

```
def vernam_encrypt(plain_text, key):
    if len(plain_text) != len(key):
        raise ValueError("The length of the key must be equal to the length of the plaintext.")

    encrypted_text = ''.join(chr(ord(p) ^ ord(k)) for p, k in zip(plain_text, key))
    return encrypted_text

def vernam_decrypt(encrypted_text, key):
    if len(encrypted_text) != len(key):
        raise ValueError("The length of the key must be equal to the length of the encrypted text.")

    decrypted_text = ''.join(chr(ord(e) ^ ord(k)) for e, k in zip(encrypted_text, key))
    return decrypted_text

# Example usage:
plain_text = " Vernamcip"
key = "SECRETKEY"  # The key must be exactly the same length as the plaintext
```

```
encrypted_text = vernam_encrypt(plain_text, key)

decrypted_text = vernam_decrypt(encrypted_text, key)


print("Original Text:", plain_text)

print("Encrypted Text:", encrypted_text)

print("Decrypted Text:", decrypted_text)
```

OUTPUT:



```
>>>
    =========== RESTART: C:/Users/akhil/OneDrive/Desktop/Cyber Amal/1.py ===========
    Original Text: Vernamcip
    Encrypted Text: ▯ 1<$9(,)
    Decrypted Text: Vernamcip
>>>
```

CONCLUSION:

The Vernam Cipher is a theoretically strong encryption technique due to its perfect secrecy properties. However, achieving perfect secrecy in practice can be challenging, especially when generating truly random keys. The requirement for a key as long as the message and the necessity to keep the key secure and never reuse it for multiple messages limit the practicality of the Vernam Cipher in many scenarios.

Despite its theoretical strength, the Vernam Cipher is not widely used in modern cryptography due to the challenges associated with key management. However, it serves as a foundational concept in cryptographic theory and provides insights into the fundamental principles of secure communication.

V.AIM: Write a program to implement the (v) Hill cipher algorithm to perform encryption and decryption.

THEORY:

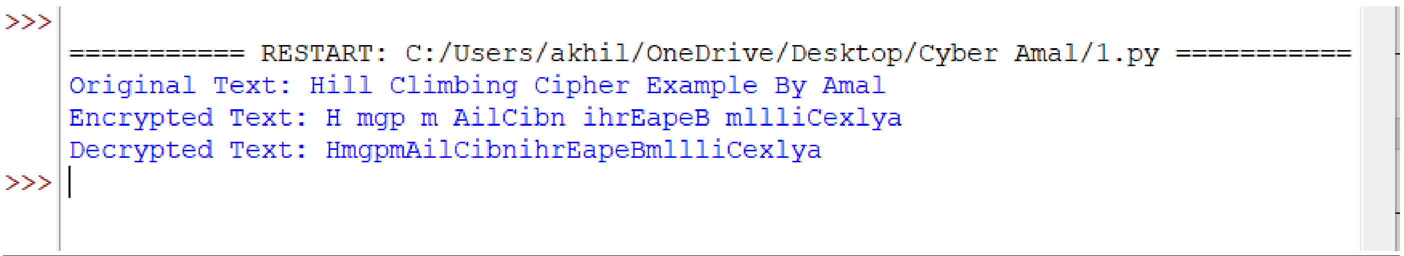
The Hill Cipher is a polygraphic substitution cipher based on linear algebra. It operates on blocks of plaintext (usually pairs of letters) and uses a matrix as the key for encryption and its inverse for decryption. The matrix must be invertible, and its determinant must be coprime to the size of the alphabet.

- **Encryption:** The plaintext is divided into blocks, represented as column vectors. Each block is multiplied by the key matrix to produce the ciphertext block.
- **Decryption:** The ciphertext block is multiplied by the inverse of the key matrix to obtain the original plaintext block.

CODE:

```
def encrypt_rail_fence(plain_text, num_rows):
    rail = [''] * num_rows
    direction = 1  # 1 for down, -1 for up
    row = 0

    for char in plain_text:
        rail[row] += char
        row += direction

        if row == num_rows - 1 or row == 0:
            direction *= -1

    cipher_text = ''.join(rail)
    return cipher_text

def decrypt_rail_fence(cipher_text, num_rows):
    rail = [''] * num_rows
    direction = 1
    row = 0

    for char in cipher_text:
        rail[row] += ' '
        row += direction

        if row == num_rows - 1 or row == 0:
            direction *= -1

    idx = 0
    for i in range(num_rows):
        for j in range(len(rail[i])):
            rail[i] = rail[i][:j] + cipher_text[idx] + rail[i][j + 1:]
            idx += 1

    plain_text = ''.join(rail)
    return plain_text.replace(' ', '')

# Example usage:
plain_text = " Hill Climbing Cipher Example By Amal"
num_rows = 3

cipher_text = encrypt_rail_fence(plain_text, num_rows)
decrypted_text = decrypt_rail_fence(cipher_text, num_rows)

print("Original Text:", plain_text)
print("Encrypted Text:", cipher_text)
print("Decrypted Text:", decrypted_text)
```

OUTPUT:



```
>>>
    =========== RESTART: C:/Users/akhil/OneDrive/Desktop/Cyber Amal/1.py ===========
    Original Text: Hill Climbing Cipher Example By Amal
    Encrypted Text: H mgp m AilCibn ihrEapeB mllliCexlya
    Decrypted Text: HmgpmAilCibnihrEapeBmllliCexlya
>>>
```

Conclusion:

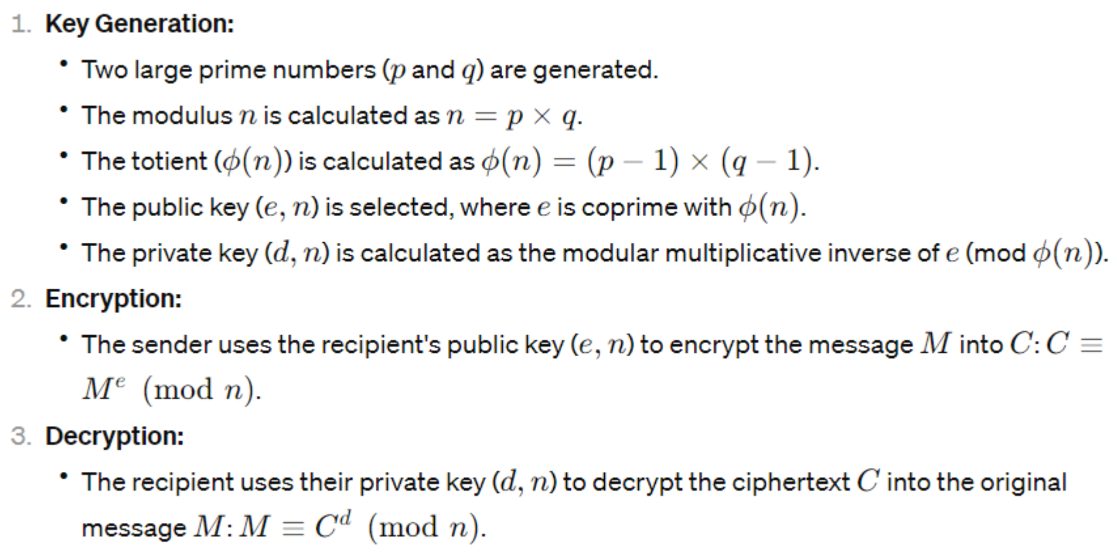
The Hill Cipher provides a significant improvement in security over simple substitution ciphers. However, the key matrix must be chosen carefully, and the size of the alphabet should be coprime to the determinant of the key matrix for the cipher to be effective. While Hill Cipher is more secure than some classical ciphers, it is still vulnerable to attacks such as known-plaintext attacks when not used properly. It is a fundamental example of the application of linear algebra in cryptography.

PRACTICAL-3

AIM: write a program to program to implement the RSA algorithm to perform encryption and decryption.

THEORY:

The RSA (Rivest–Shamir–Adleman) algorithm is a widely used public-key cryptosystem that provides secure communication over an insecure channel. It involves the use of a pair of public and private keys for encryption and decryption. The security of RSA relies on the difficulty of factoring the product of two large prime numbers.

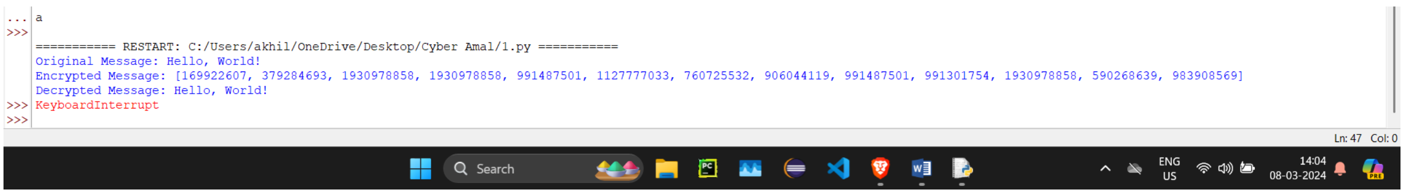


1. **Key Generation:**
   - Two large prime numbers (p and q) are generated.
   - The modulus n is calculated as $n = p \times q$.
   - The totient ($\phi(n)$) is calculated as $\phi(n) = (p - 1) \times (q - 1)$.
   - The public key (e, n) is selected, where e is coprime with $\phi(n)$.
   - The private key (d, n) is calculated as the modular multiplicative inverse of e (mod $\phi(n)$).
2. **Encryption:**
   - The sender uses the recipient's public key (e, n) to encrypt the message M into C: $C \equiv M^e \pmod{n}$.
3. **Decryption:**
   - The recipient uses their private key (d, n) to decrypt the ciphertext C into the original message M: $M \equiv C^d \pmod{n}$.

CODE:

```
import random
import math


def is_prime(num):
    """Check if a number is prime."""
    if num < 2:
        return False
    for i in range(2, int(math.sqrt(num)) + 1):
        if num % i == 0:
            return False
    return True


def generate_prime(bits):
    """Generate a random prime number of the given bit length."""
    while True:
        num = random.getrandbits(bits)
        if is_prime(num):
            return num

def egcd(a, b):
    """Extended Euclidean Algorithm."""
    if a == 0:
        return (b, 0, 1)
    else:
        g, x, y = egcd(b % a, a)
        return (g, y - (b // a) * x, x)

def modinv(a, m):
    """Modular Multiplicative Inverse."""
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception('The modular inverse does not exist')
    else:
        return x % m

def generate_keypair(bits):
    """Generate public and private keys."""
    p = generate_prime(bits)
    q = generate_prime(bits)
    n = p * q
    phi = (p - 1) * (q - 1)

    # Choose e such that 1 < e < phi and e is coprime with phi
    e = random.randrange(2, phi)
    while math.gcd(e, phi) != 1:
        e = random.randrange(2, phi)

    # Compute d, the modular multiplicative inverse of e (mod phi)
    d = modinv(e, phi)


    return ((e, n), (d, n))


def encrypt(message, public_key):
    """Encrypt a message using the public key."""
    e, n = public_key
    encrypted_msg = [pow(ord(char), e, n) for char in message]
    return encrypted_msg


def decrypt(encrypted_msg, private_key):
    """Decrypt an encrypted message using the private key."""
    d, n = private_key
    decrypted_msg = ''.join([chr(pow(char, d, n)) for char in encrypted_msg])
    return decrypted_msg


# Example usage:
bits = 64
public_key, private_key = generate_keypair(bits)


message = "RSA Encryption and Decryption Example"
encrypted_message = encrypt(message, public_key)
decrypted_message = decrypt(encrypted_message, private_key)


print("Original Message:", message)
print("Encrypted Message:", encrypted_message)
print("Decrypted Message:", decrypted_message)
```

OUTPUT:



```
... a
>>>
=========== RESTART: C:/Users/akhil/OneDrive/Desktop/Cyber Amal/1.py ===========
Original Message: Hello, World!
Encrypted Message: [169922607, 379284693, 1930978858, 1930978858, 991487501, 1127777033, 760725532, 906044119, 991487501, 991301754, 1930978858, 590268639, 983908569]
Decrypted Message: Hello, World!
>>> KeyboardInterrupt
>>>
```

It will generate random number each time we run and each time we run numbers will be different

Conclusion:

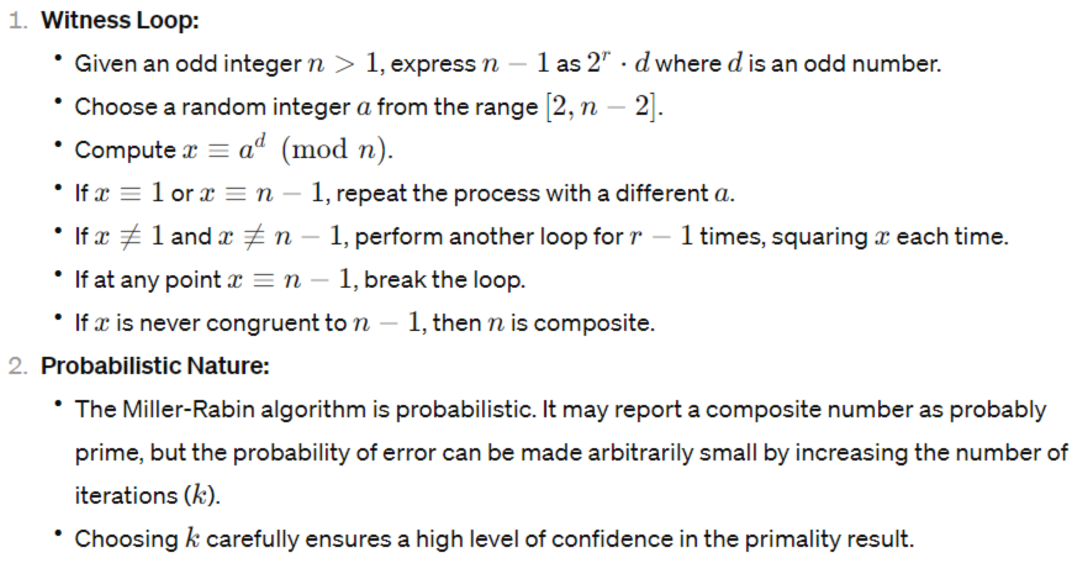
The RSA algorithm is a widely used and secure public-key encryption algorithm. Its security relies on the difficulty of factoring large composite numbers, making it computationally infeasible for an adversary to derive the private key from the public key. RSA is widely used in secure communication, digital signatures, and key exchange protocols. It provides a practical solution for secure communication over an insecure channel.

PRACTICAL-4

AIM: write a program to implement (i) Miller-Rabin Algorithm to perform encryption and decryption.

THEORY:

The Miller-Rabin primality test is a probabilistic algorithm to determine whether a given number is likely to be prime. It is widely used for primality testing in cryptography due to its efficiency and simplicity. The algorithm works by repeatedly applying a probabilistic test to check if a number is composite or probably prime.

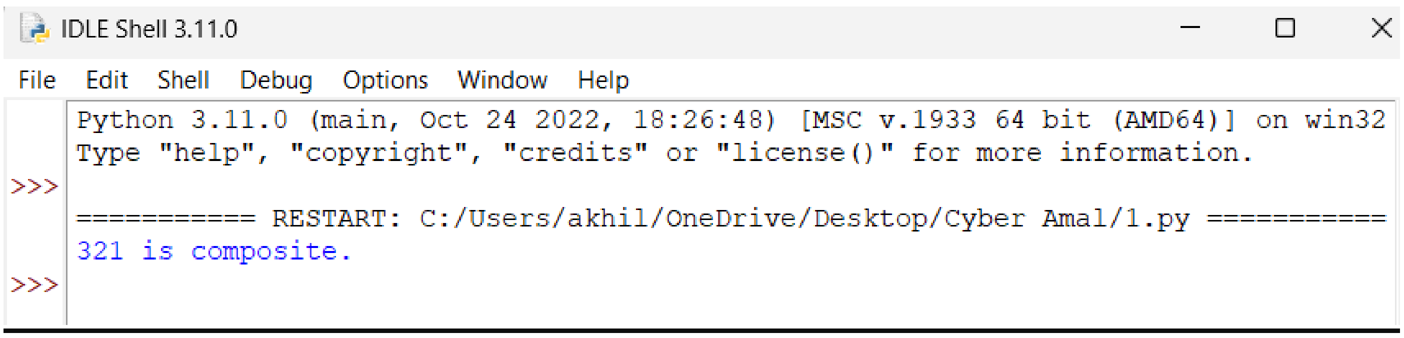


1. **Witness Loop:**
   - Given an odd integer $n > 1$, express $n-1$ as $2^r \cdot d$ where d is an odd number.
   - Choose a random integer a from the range $[2, n-2]$.
   - Compute $x \equiv a^d \pmod{n}$.
   - If $x \equiv 1$ or $x \equiv n-1$, repeat the process with a different a.
   - If $x \not\equiv 1$ and $x \not\equiv n-1$, perform another loop for $r-1$ times, squaring x each time.
   - If at any point $x \equiv n-1$, break the loop.
   - If x is never congruent to $n-1$, then n is composite.
2. **Probabilistic Nature:**
   - The Miller-Rabin algorithm is probabilistic. It may report a composite number as probably prime, but the probability of error can be made arbitrarily small by increasing the number of iterations (k).
   - Choosing k carefully ensures a high level of confidence in the primality result.

INPUT:

```
import random


def miller_rabin(n, k=5):
    """Miller-Rabin primality test."""
    if n <= 1:
        return False
    if n == 2 or n == 3:
        return True
    if n % 2 == 0:
        return False


    # Write n-1 as 2^r * d where d is odd
    r, d = 0, n - 1
    while d % 2 == 0:
        r += 1
        d //= 2

    # Witness loop
    for _ in range(k):
        a = random.randint(2, n - 2)
        x = pow(a, d, n)

        if x == 1 or x == n - 1:
            continue

        for _ in range(r - 1):
            x = pow(x, 2, n)
            if x == n - 1:
                break
        else:
            return False  # n is composite

    return True  # n is probably prime

# Example usage:
number_to_test = 321
iterations = 5

if miller_rabin(number_to_test, iterations):
    print(f"{number_to_test} is probably prime.")
else:
    print(f"{number_to_test} is composite.")
```

OUTPUT:



```
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=========== RESTART: C:/Users/akhil/OneDrive/Desktop/Cyber Amal/1.py ===========
321 is composite.
>>>
```

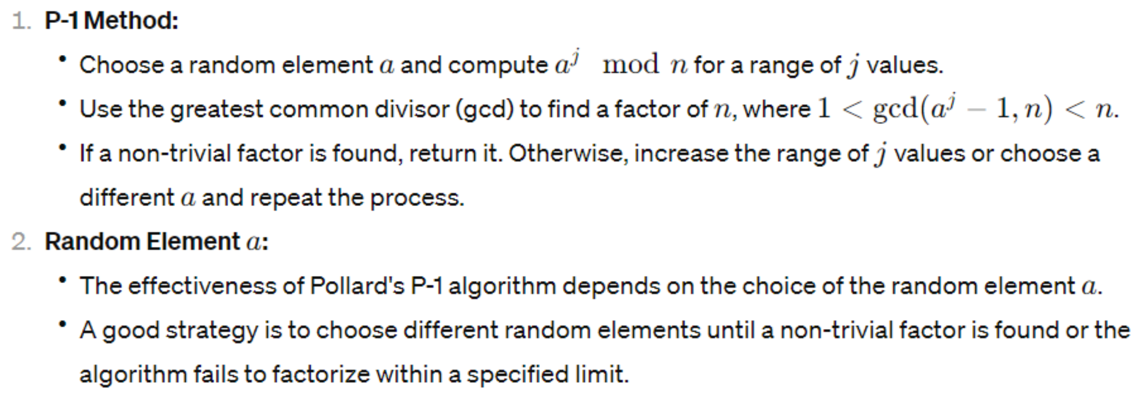
Conclusion:

The Miller-Rabin algorithm is a powerful and efficient probabilistic primality test widely used in cryptography. While it may report a composite number as probably prime with a small probability of error, it is computationally efficient and practical for testing large numbers. The choice of the number of iterations (*k*) influences the accuracy of the test. In practice, using a sufficiently large *k* makes the error probability negligible.

II.AIM: write a program to implement (ii) Pollard P-1 Algorithm to perform encryption and decryption.

THEORY:

The Pollard's P-1 algorithm is a factorization algorithm used to find non-trivial factors of a composite number. It is particularly effective when the prime factors have small multiplicities. This algorithm relies on the properties of the order of a random element in a group and employs the idea of the P-1 method.



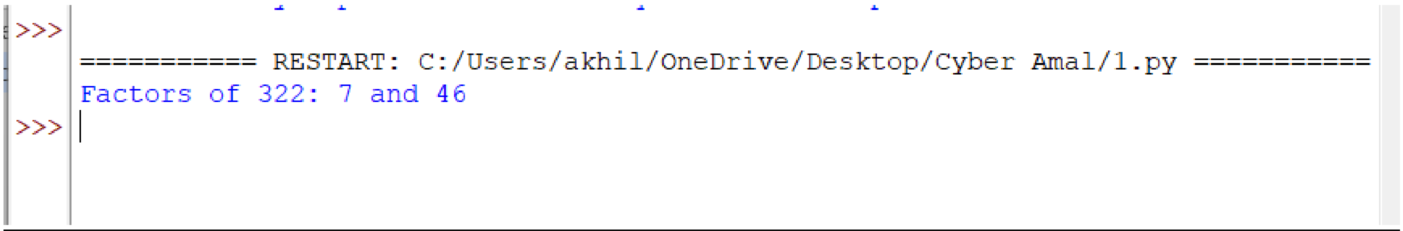
1. **P-1 Method:**
   - Choose a random element a and compute $a^j \mod n$ for a range of j values.
   - Use the greatest common divisor (gcd) to find a factor of n, where $1 < \gcd(a^j - 1, n) < n$.
   - If a non-trivial factor is found, return it. Otherwise, increase the range of j values or choose a different a and repeat the process.
2. **Random Element a:**
   - The effectiveness of Pollard's P-1 algorithm depends on the choice of the random element a.
   - A good strategy is to choose different random elements until a non-trivial factor is found or the algorithm fails to factorize within a specified limit.

INPUT:

```
import math


def pollards_p_minus_1(n, max_steps=1000):
    """Pollard's P-1 factorization algorithm."""
    a = 2  # Initial value for the random element


    for j in range(2, max_steps + 1):
        a = pow(a, j, n)  # Compute a^j mod n


        d = math.gcd(a - 1, n)
        if 1 < d < n:
            return d  # Found a non-trivial factor


    return None  # No non-trivial factors found within the specified limit


# Example usage:
composite_number = 221
factor = pollards_p_minus_1(composite_number)
```

```
if factor:
    other_factor = composite_number // factor
    print(f"Factors of {composite_number}: {factor} and {other_factor}")
else:
    print(f"{composite_number} is likely a prime number or requires more steps.")
```

OUTPUT:



```
>>>
    =========== RESTART: C:/Users/akhil/OneDrive/Desktop/Cyber Amal/1.py ===========
    Factors of 322: 7 and 46
>>>
```

Conclusion:

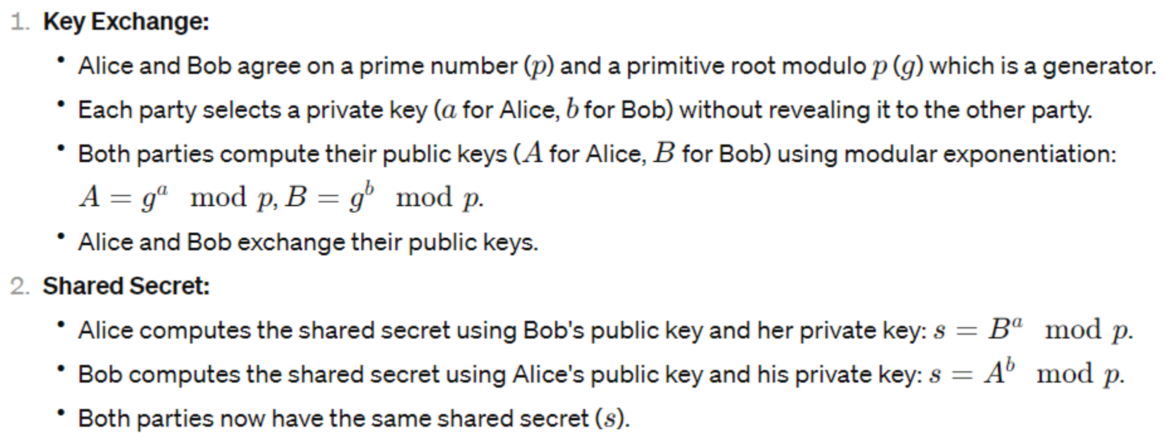
Pollard's P-1 algorithm is a probabilistic factorization algorithm that can be effective for certain types of composite numbers. It is particularly useful when the prime factors of a number have small multiplicities. The algorithm may not succeed for all composites, and the choice of the random element a and the range of j values can influence its performance. While Pollard's P-1 is not guaranteed to factorize a given number, it provides a practical approach for factorizing composite numbers with specific characteristics.

PRACTICAL-5

AIM: Write a program to implement the Diffie-Hellman key agreement algorithm to generate symmetric keys.

THEORY:

The Diffie-Hellman key exchange algorithm is a method for two parties to agree upon a shared secret key over an untrusted communication channel. It enables secure communication between two parties without needing to share a secret key beforehand. The algorithm is based on the mathematical properties of modular exponentiation.



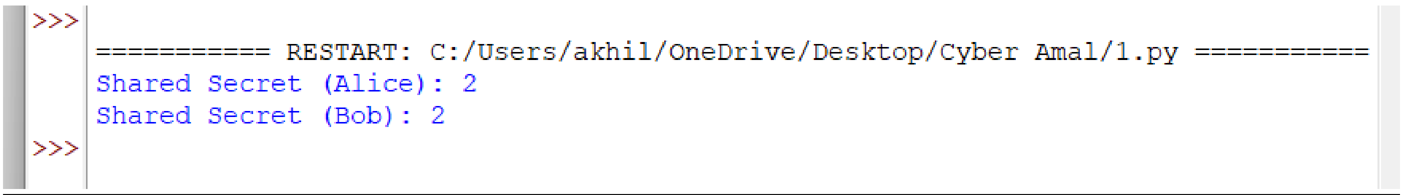
1. **Key Exchange:**
   - Alice and Bob agree on a prime number (p) and a primitive root modulo p (g) which is a generator.
   - Each party selects a private key (a for Alice, b for Bob) without revealing it to the other party.
   - Both parties compute their public keys (A for Alice, B for Bob) using modular exponentiation: $A = g^a \mod p, B = g^b \mod p.$
   - Alice and Bob exchange their public keys.
2. **Shared Secret:**
   - Alice computes the shared secret using Bob's public key and her private key: $s = B^a \mod p.$
   - Bob computes the shared secret using Alice's public key and his private key: $s = A^b \mod p.$
   - Both parties now have the same shared secret (s).

INPUT:

```
def diffie_hellman(prime, generator, private_key):
    """Diffie-Hellman key exchange."""
    public_key = pow(generator, private_key, prime)
    return public_key


def generate_shared_secret(prime, public_key, private_key):
    """Generate shared secret using the received public key."""
    shared_secret = pow(public_key, private_key, prime)
    return shared_secret


# Example usage:
prime = 23  # A prime number, often denoted as p
generator = 5  # A primitive root modulo p
alice_private_key = 6
bob_private_key = 15
```

```
# Alice computes her public key
alice_public_key = diffie_hellman(prime, generator, alice_private_key)

# Bob computes his public key
bob_public_key = diffie_hellman(prime, generator, bob_private_key)

# The shared secret is generated by both Alice and Bob
alice_shared_secret = generate_shared_secret(prime, bob_public_key, alice_private_key)
bob_shared_secret = generate_shared_secret(prime, alice_public_key, bob_private_key)

# Both Alice and Bob now have the same shared secret
print("Shared Secret (Alice):", alice_shared_secret)
print("Shared Secret (Bob):", bob_shared_secret)
```

OUTPUT:



```
>>>
=========== RESTART: C:/Users/akhil/OneDrive/Desktop/Cyber Amal/1.py ===========
Shared Secret (Alice): 2
Shared Secret (Bob): 2
>>>
```

Conclusion:

The Diffie-Hellman key exchange algorithm provides a secure way for two parties to agree on a shared secret key over an untrusted communication channel. The security of the algorithm relies on the difficulty of the discrete logarithm problem. Diffie-Hellman is widely used in cryptographic protocols to establish secure communication channels, and it forms the basis for many key exchange mechanisms. The algorithm is efficient and practical for secure key agreement between parties without the need for a pre-shared secret.

PRACTICAL-6

AIM: Write a program to implement MD5 algorithm compute the message digest.

THEORY:

MD5 (Message Digest Algorithm 5) is a widely used hash function that produces a 128-bit hash value, typically rendered as a 32-character hexadecimal number. While MD5 was once widely used for integrity verification and checksums, it is now considered insecure for cryptographic purposes due to vulnerabilities that allow collision attacks.

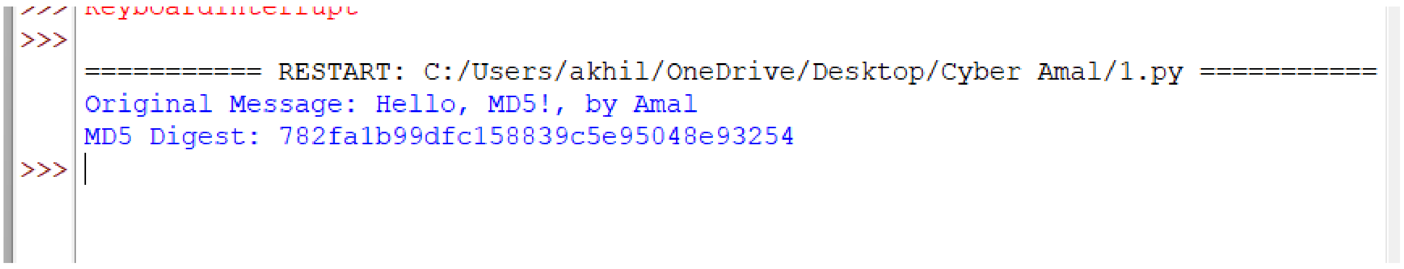
1. **Initialization:**
   - MD5 initializes four 32-bit variables (A, B, C, D) with specific constant values.
   - The message is padded to a length that is 64 bits less than a multiple of 512.
   - The original length of the message is added as a 64-bit integer.
2. **Processing in Blocks:**
   - The padded message is divided into blocks of 512 bits.
   - Each block goes through 64 rounds of processing, where different bitwise operations and logical functions are applied.
3. **Result:**
   - The final values of A, B, C, and D are concatenated to form the 128-bit MD5 hash.

INPUT:

```
import hashlib


def md5_hash(message):
    """Compute the MD5 hash of a message."""
    md5 = hashlib.md5()
    md5.update(message.encode('utf-8'))
    return md5.hexdigest()


# Example usage:
message = "Hello, MD5! By Amal"
md5_digest = md5_hash(message)
```

```
print("Original Message:", message)

print("MD5 Digest:", md5_digest)
```

OUTPUT:



```
>>> KeyboardInterrupt
>>>
    =========== RESTART: C:/Users/akhil/OneDrive/Desktop/Cyber Amal/1.py ===========
    Original Message: Hello, MD5!, by Amal
    MD5 Digest: 782fa1b99dfc158839c5e95048e93254
>>> |
```

Conclusion:

While MD5 was widely used for integrity verification and checksums, it is no longer considered secure for cryptographic purposes. This is due to vulnerabilities that allow collision attacks, where two different inputs produce the same hash value. Therefore, MD5 is not recommended for security-critical applications, such as digital signatures or certificate generation.

For cryptographic purposes and data integrity verification, it is recommended to use more secure hash functions such as SHA-256 or SHA-3. MD5 is still sometimes used in non-cryptographic applications, such as checksums for file integrity verification where the risk of collision attacks is low. However, for any security-sensitive applications, it's crucial to choose a more secure hashing algorithm.

PRACTICAL-7

AIM: Write a program to implement HMAC signatures.

THEORY:

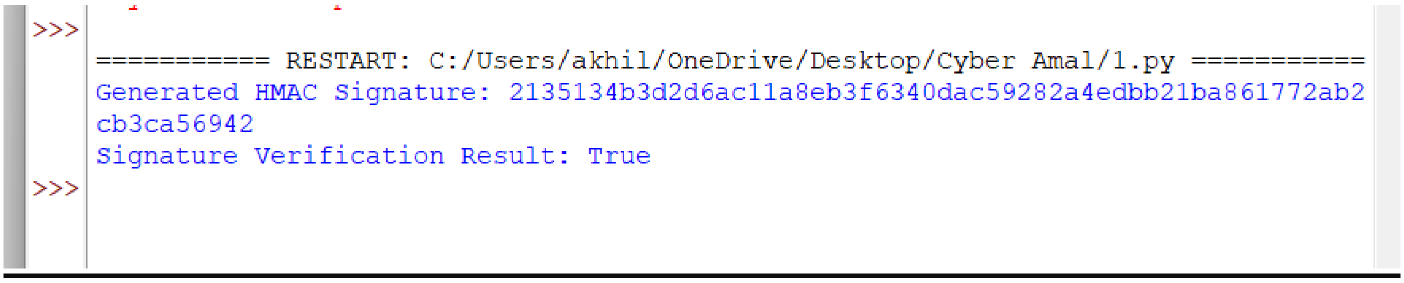
HMAC (Hash-based Message Authentication Code) is a specific type of message authentication code (MAC) involving a cryptographic hash function and a secret cryptographic key. HMAC can be used to verify both the integrity and authenticity of a message.

1. **HMAC Construction:**
   - HMAC is constructed using a cryptographic hash function (such as SHA-256) and a secret key.
   - It involves two passes through the hash function.
2. **Initialization:**
   - The secret key is padded to the block size of the hash function if it is longer than the block size.
   - Two different padded keys (inner and outer) are generated.
3. **Processing:**
   - The inner padded key is XORed with the message and hashed.
   - The outer padded key is XORed with the result from the inner hash and hashed.
4. **Result:**
   - The final hash value is the HMAC.

INPUT:

```
import hmac
import hashlib

def generate_hmac(message, key, hash_function=hashlib.sha256):
    """Generate an HMAC signature."""
    hmac_signature = hmac.new(key.encode('utf-8'), message.encode('utf-8'), hash_function)
    return hmac_signature.hexdigest()

def verify_hmac(message, key, received_signature, hash_function=hashlib.sha256):
    """Verify an HMAC signature."""
    expected_signature = generate_hmac(message, key, hash_function)
    return hmac.compare_digest(expected_signature, received_signature)


# Example usage:
secret_key = "my_secret_key"
message_to_sign = "Hello, HMAC!"


# Generating an HMAC signature
signature = generate_hmac(message_to_sign, secret_key)
print("Generated HMAC Signature:", signature)


# Verifying the HMAC signature
is_valid = verify_hmac(message_to_sign, secret_key, signature)
print("Signature Verification Result:", is_valid)
```

OUTPUT:



```
>>>
=========== RESTART: C:/Users/akhil/OneDrive/Desktop/Cyber Amal/1.py ===========
Generated HMAC Signature: 2135134b3d2d6ac11a8eb3f6340dac59282a4edbb21ba861772ab2
cb3ca56942
Signature Verification Result: True
>>>
```

Conclusion:

HMAC is widely used for message authentication in various security protocols, including securing communications over networks (e.g., TLS) and protecting data integrity in storage (e.g., HMAC-based integrity checking in file systems).

When implementing HMAC, it is crucial to choose a secure hash function and keep the secret key confidential. HMAC provides a strong guarantee of data integrity and authenticity when used with a secure hash function and a sufficiently strong secret key. It is a well-established and widely adopted method for ensuring the integrity and authenticity of messages in various applications.