**Emerging Blockchain Models for Digital Currencies**

Name: Shaik Aarif

Regd No: 2200033144

Exp 1: Implementation of the Custom Symmetric Key Encryption Algorithm.

**Description of the Algorithm:**

1. **Key Generation (GenKey)**:
   * The key is calculated by summing the ASCII values of all characters in the plaintext string.
   * This serves as the basis for encryption and decryption.
2. **Simple XOR Cipher (Encrypt1 and Decrypt1)**:
   * Each character of the plaintext is XOR-ed with the key to produce the ciphertext.
   * Decryption reverses this process using the same key.
3. **Enhanced XOR Cipher with Multiplicative Factor (Encrypt2 and Decrypt2)**:
   * Each character of the plaintext is XOR-ed with the product of the key and a position-based multiplier (j).
   * If the resulting value exceeds the Unicode limit (1114111), it is wrapped around using modulo operation.
   * This adds an additional layer of complexity, making it harder to decode without the exact key and logic.
4. **Encryption Process**:
   * Encrypt1 applies a simple XOR operation.
   * Encrypt2 uses a position-dependent multiplier (j) to modify the XOR operation for each character.
5. **Decryption Process**:
   * The decryption process mirrors the encryption logic, ensuring that the ciphertext is reverted to plaintext using the same key and algorithm.
6. **Output**:
   * The program generates two versions of ciphertext: one with a normal XOR logic (Encrypt1) and another with a custom XOR logic (Encrypt2).
   * It then decrypts both ciphertexts back to plaintext to verify the correctness of the algorithms.
7. **Boundary Handling in Encrypt2 and Decrypt2**:
   * If the calculated character value exceeds the Unicode limit, it wraps around using % 1114112.
   * This ensures compatibility with valid Unicode character ranges.
8. **Testing**:
   * The program prints intermediate values for debugging (value during encryption and decryption in Encrypt2).
   * It displays both the encrypted and decrypted outputs for comparison.

Code:

1. def GenKey(plainText):

2.     key = 0

3.     for i in plainText:

4.         key += ord(i)

5.     return key

6.

7. def Encrypt1(plainText, key):

8.     cipherText = ""

9.     for i in plainText:

10.         cipherText += chr(ord(i) ^ key)

11.     return cipherText

12.

13. def Decrypt1(cipherText, key):

14.     plainText = ""

15.     for i in cipherText:

16.         plainText += chr(ord(i) ^ key)

17.     return plainText

18.

19. def Encrypt2(plainText, key):

20.     cipherText = ""

21.     j=1

22.     for i in plainText:

23.         value = ord(i) ^ key \* j

24.         if value > 1114111:

25.             value = value % 1114112

26.         print(value , end=' ')

27.         cipherText += chr(value)

28.         j+=1

29.     return cipherText

30.

31. def Decrypt2(cipherText, key):

32.     plainText = ""

33.     j=1

34.     for i in cipherText:

35.         value = ord(i) ^ key \* j

36.         if value > 1114111:

37.             value = value % 1114112

38.         print(value , end=' ')

39.         plainText += chr(value)

40.         j+=1

41.     return plainText

42.

43. plainText = "Emerging BlockChain"

44. key=GenKey(plainText)

45. cipherText = Encrypt1(plainText, key)

46. DecryptText = Decrypt1(cipherText, key)

47.

48. cipherText1 = Encrypt2(plainText, key)

49. DecryptText1 = Decrypt2(cipherText1, key)

50.

51. print("Key: ", key)

52. print("\n")

53. print("Normal XOR Cipher Text: ", cipherText)

54. print("Normal XOR Decrypt Text: ", DecryptText)

55.

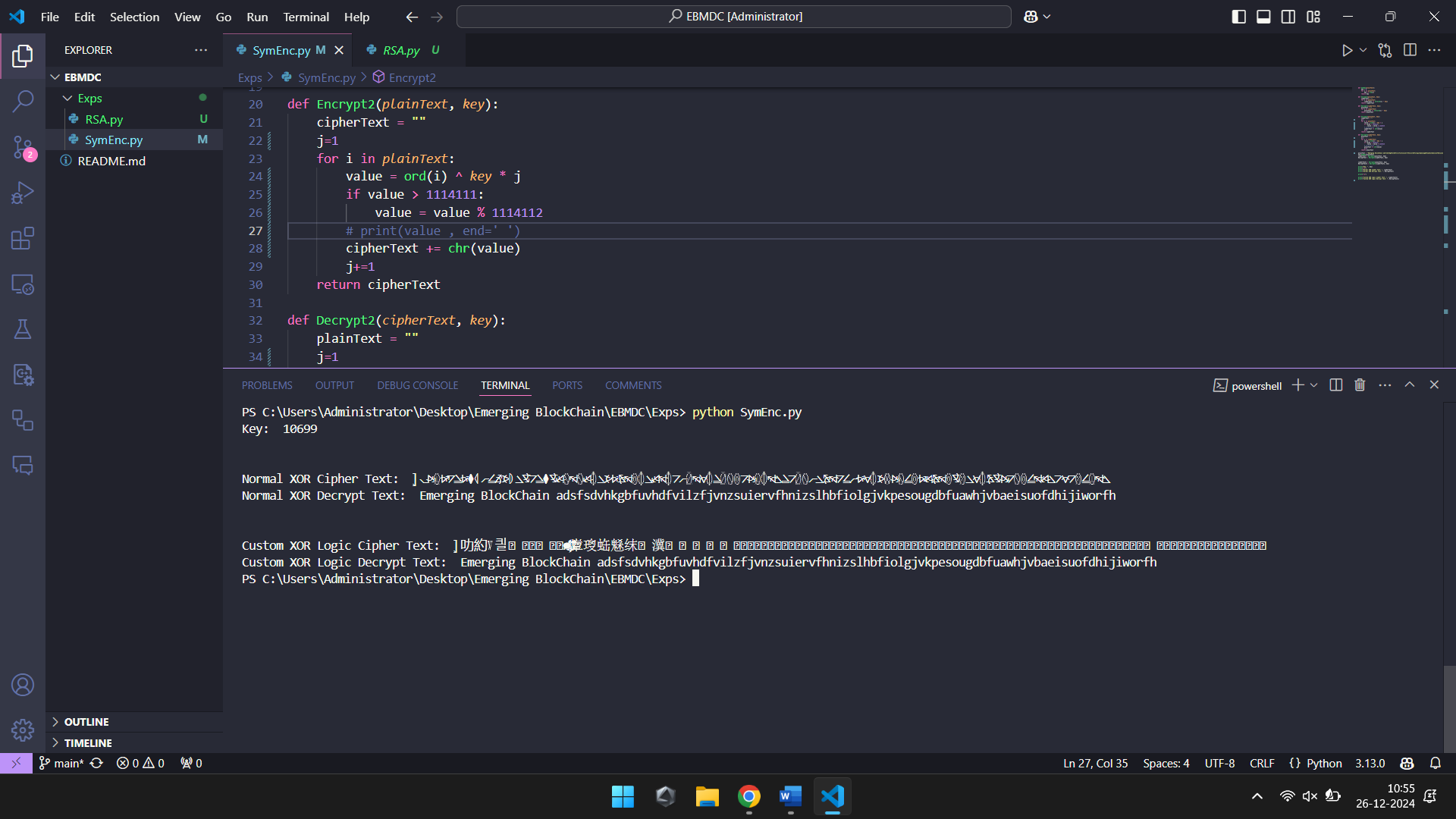
56. print("\n")

57.

58. print("Custom XOR Logic Cipher Text: ", cipherText1)

59. print("Custom XOR Logic Decrypt Text: ", DecryptText1)

Execution: (Screenshot)



Exp 2: Implementation of the RSA Encryption Algorithm with Key Generation.

**Description of the RSA Algorithm Implementation:**

1. **Purpose**:
   * Implements the RSA encryption and decryption process to securely encode and decode messages.

**Steps in the Algorithm:**

1. **Key Components**:
   * **Public Key**: (e, n) used for encryption.
   * **Private Key**: (d, n) used for decryption.
2. **Prime Number Generation**:
   * generate\_prime\_candidate(length): Generates a random odd number within the range specified by the bit length.
   * is\_prime(n): Validates whether a number is prime using trial division up to the square root of the number.
   * generate\_prime\_number(length): Continually generates and validates random primes of the specified bit length.
3. **Key Generation (generate\_keys)**:
   * Two distinct large prime numbers, p and q, are generated.
   * The modulus n is calculated as n = p×q ,.
   * Euler's totient function is computed as ϕ=(p−1)×(q−1).
   * Public exponent e is chosen such that 1<e<ϕ and gcd(e,ϕ)=1.
   * Private exponent d is calculated as the modular inverse of e modulo phi.
4. **Encryption (encrypt)**:
   * Converts the plaintext characters to their Unicode code points.
   * Each code point is raised to the power of e (public exponent) modulo n.
   * Produces the ciphertext as a list of integers.
5. **Decryption (decrypt)**:
   * Each ciphertext integer is raised to the power of d (private exponent) modulo n.
   * Converts the resulting values back to characters, forming the original plaintext.

**Key Features:**

1. **Security**:
   * The security of RSA relies on the difficulty of factoring the product of two large prime numbers.
   * Only the private key holder can decrypt the ciphertext.
2. **Randomness**:
   * Prime numbers p and q are randomly generated for each key pair.
   * This ensures unique and secure keys for every execution.
3. **Modular Arithmetic**:
   * The algorithm heavily uses modular arithmetic for key generation, encryption, and decryption, which is efficient and secure.

**Example Walkthrough:**

* **Key Generation**:
  + Random primes p and q are selected (e.g., p = 211, q = 223 for an 8-bit size).
  + n=211×223=47053 , ϕ=(211−1)×(223−1)=46620.
  + e is chosen (e.g., e = 13 such that gcd(13, 46620) = 1).
  + d is computed as e−1 mod  ϕ (e.g., d = 35957).
* **Encryption**:
  + Message: "Hello Aarif".
  + Convert each character to ciphertext: cipher=ord(char)emod  n.
* **Decryption**:
  + Convert each ciphertext value back to plaintext: plain=cipherdmod.

**Output:**

* Displays public and private keys.
* Shows the encrypted and decrypted message, verifying the correctness of the implementation.

Code:

1. import random

2.

3. def gcd(a, b):

4.     while b != 0:

5.         a, b = b, a % b

6.     return a

7.

8. def mod\_inverse(e, phi):

9.     d, x1, x2, y1 = 0, 0, 1, 1

10.     phi0 = phi

11.     while e > 0:

12.         temp1, temp2 = phi // e, phi % e

13.         x = x2 - temp1 \* x1

14.         y = d - temp1 \* y1

15.         phi, e = e, temp2

16.         x2, x1 = x1, x

17.         d, y1 = y1, y

18.     if phi == 1:

19.         return d + phi0

20.

21. def is\_prime(n):

22.     if n <= 1:

23.         return False

24.     for i in range(2, int(n\*\*0.5) + 1):

25.         if n % i == 0:

26.             return False

27.     return True

28.

29. def generate\_prime\_candidate(length):

30.     p = random.randrange(2\*\*(length - 1), 2\*\*length)

31.     if p % 2 == 0:

32.         p += 1

33.     return p

34.

35. def generate\_prime\_number(length=8):

36.     p = 4

37.     while not is\_prime(p):

38.         p = generate\_prime\_candidate(length)

39.     return p

40.

41. def generate\_keys(keysize=8):

42.     p = generate\_prime\_number(keysize)

43.     q = generate\_prime\_number(keysize)

44.     while p == q:

45.         q = generate\_prime\_number(keysize)

46.     n = p \* q

47.     phi = (p - 1) \* (q - 1)

48.

49.     e = random.randrange(1, phi)

50.     while gcd(e, phi) != 1:

51.         e = random.randrange(1, phi)

52.

53.     d = mod\_inverse(e, phi)

54.

55.     return (e, n), (d, n)

56.

57. def encrypt(public\_key, plaintext):

58.     e, n = public\_key

59.     ciphertext = [pow(ord(char), e, n) for char in plaintext]

60.     return ciphertext

61.

62. def decrypt(private\_key, ciphertext):

63.     d, n = private\_key

64.     plaintext = ''.join([chr(pow(char, d, n)) for char in ciphertext])

65.     return plaintext

66.

67. public\_key, private\_key = generate\_keys(keysize=8)

68. print("Public key:", public\_key)

69. print("Private key:", private\_key)

70.

71. message = "Hello Aarif"

72. print("\nOriginal message:", message)

73.

74. ciphertext = encrypt(public\_key, message)

75. print("\nEncrypted message:", ciphertext)

76.

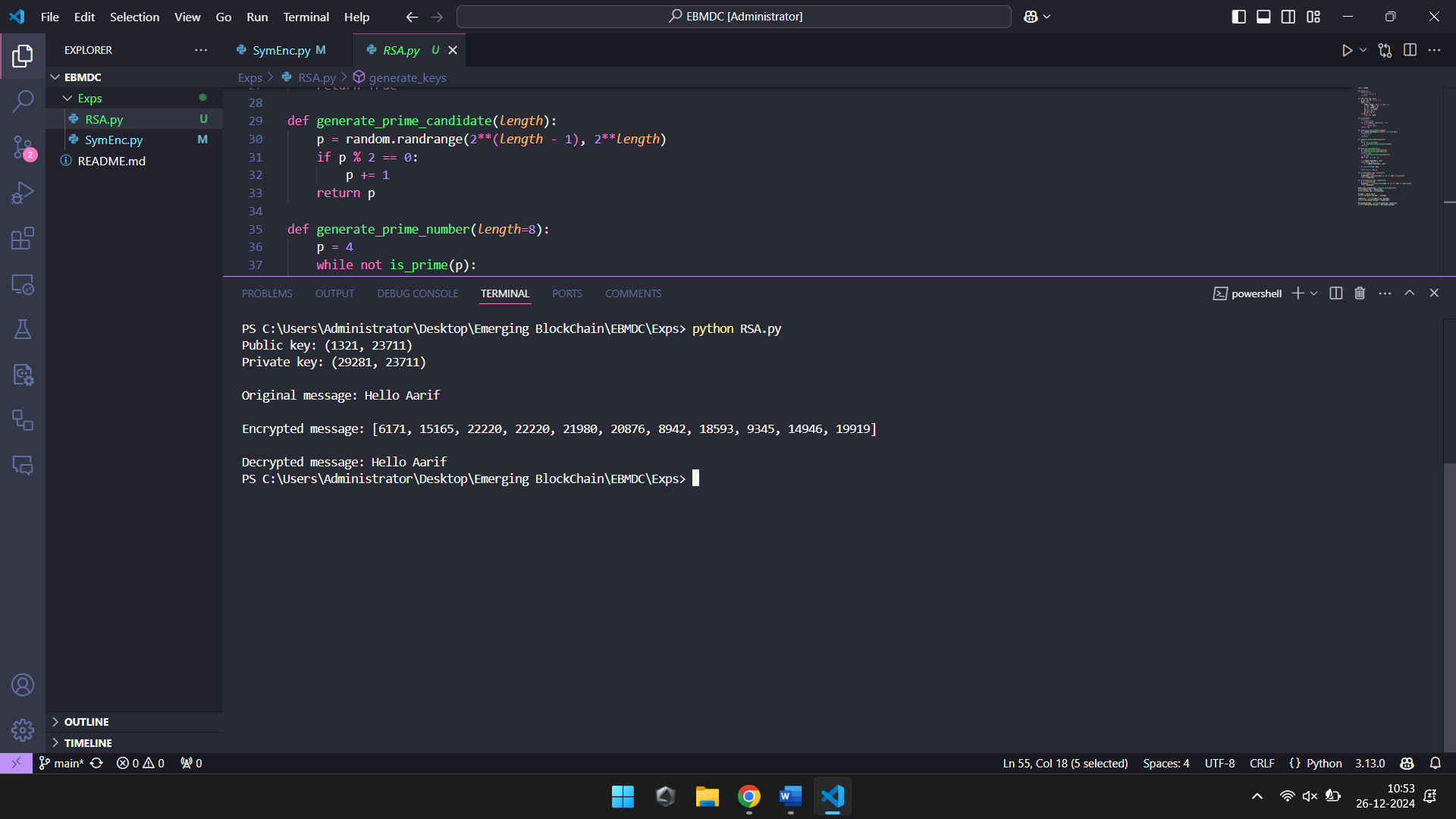
77. decrypted\_message = decrypt(private\_key, ciphertext)

78. print("\nDecrypted message:", decrypted\_message)

79.

80.

Execution: (Screenshot)



Exp 3: Implementation of the Custom Asymmetric Key Encryption Algorithm.

**Description:**

**Key Generation**

* The key\_generation function generates a public-private key pair using predefined values for prime numbers p and q.
* n is the product of p and q, and g is a primitive root modulo n.
* The private key is a (a secret integer), and the public key consists of g, A (where A = g^a mod n), and n.
* The function returns both the private key (a, n) and public key (g, A, n).

**Encryption**

* The encrypt function encrypts a message m using the public key.
* A random integer k is selected, and then two values are computed:
  + c1 = g^k mod n (first part of the ciphertext)
  + c2 = (m \* A^k) mod n (second part of the ciphertext)
* It returns the encrypted message as a tuple (c1, c2).

**Decryption**

* The decrypt function decrypts the ciphertext (c1, c2) using the private key (a, n).
* It first calculates S = c1^a mod n, which is the shared secret.
* Then, the modular inverse of S (denoted S^-1 mod n) is computed.
* Finally, the original message m is recovered as m = (c2 \* S^-1) mod n.

**Main Program**

1. **Key Generation:** It calls key\_generation to obtain the public and private keys.
2. **Message Conversion:** The original message, "BlockChain", is converted into a numeric format by converting each character to its ASCII value (minus 'A' to shift into a suitable range).
3. **Encryption:** Each numeric value is encrypted using the public key, and the encrypted pairs (c1, c2) are stored in a list res.
4. **Decryption:** The ciphertext (c1, c2) pairs are decrypted using the private key, and the numeric results are stored in resdesc.
5. **Message Reconstruction:** Finally, the decrypted numeric values are converted back to characters (by adding the ASCII value of 'A') and printed as the decrypted message.

1. from sympy import mod\_inverse

2.

3. def key\_generation():

4.

5.     p = 7

6.     q = 11

7.     n = p \* q

8.     g = 5

9.     a = 3

10.     A = pow(g, a, n)

11.

12.     private\_key = (a,n)

13.     public\_key = (g, A, n)

14.

15.     return private\_key, public\_key

16.

17. def encrypt(public\_key, m):

18.     g, A, n = public\_key

19.     k = 6

20.

21.     c1 = pow(g, k, n)

22.     c2 = (m \* pow(A, k, n)) % n

23.

24.     return c1, c2

25.

26. def decrypt(private\_key, c1, c2):

27.

28.     a,n = private\_key

29.

30.     S = pow(c1, a, n)

31.     S\_inverse = mod\_inverse(S, n)

32.

33.     m = (c2 \* S\_inverse) % n

34.

35.     return m

36.

37. private\_key, public\_key = key\_generation()

38.

39. print("Private Key:", private\_key)

40. print("Public Key (g, A, n):", public\_key)

41.

42. orginal\_message = "BlockChain"

43. marr= []

44. for i in orginal\_message:

45.     marr.append(ord(i)-ord('A'))

46. # m = 13

47. print("Original Message in number format:", marr)

48.

49. res= []

50. for m in marr:

51.     c1, c2 = encrypt(public\_key, m)

52.     res.append((c1, c2))

53.

54. print("Cipher Text: ", res)

55.

56. resdesc = []

57. for c1, c2 in res:

58.     decrypted\_message = decrypt(private\_key, c1, c2)

59.     resdesc.append(decrypted\_message)

60.

61. print("Decrypted Message:", resdesc)

62.

63. for i in resdesc:

64.     i= i+ord('A')

65.     print(chr(i), end='')

Screenshot:

