



NIST COLLEGE

Banepa, Kavre

Bachelor of Computer Science & Information Technology

Fifth Semester

Example Programs on Cryptography

Compiled by

Er. Dinesh Ghemosu

Instructions to Students

1. Students are required to bring an initial lab sheet on the lab period. An initial lab sheet should contain:
 - i. A Lab Number
 - ii. A Lab Title
 - iii. Objectives
 - iv. Theory
 - v. Algorithms
 - vi. Pseudo codes
2. Students are obliged to bring required own materials, notes, books, laptop(if you have, that will be better), pencil, pen, eraser etc.
3. After finishing the lab of that respective day, students need to bring final lab sheet which contains outputs, discussions and conclusion in next lab period. Late submission will not be entertained.
4. Students are expected to maintain discipline in the lab.
5. Students need to appear in lab exam for lab evaluation.

LAB-1

FAMILIARIZATION CLASSICAL CIPHER

Objectives:

1. To encrypt and decrypt a Ceaser cipher.
2. To perform cyptoanalysis of Ceaser cipher.
3. To encrypt and decrypt a message using Monoalphabetic cipher.
4. To encrypt and decrypt a message using Playfair technique.

Program 1

Program 1: WAP to reverse an input message.

```
1 # Reverse Cipher
2 msg = input("Input your message: ")
3 n = len(msg)
4 reverse_msg = ''
5 i = n - 1
6 while (i >= 0):
7     reverse_msg = reverse_msg + msg[i]
8     i = i-1
9 print("Revers cipher of message is: ", reverse_msg)
```

Program 2

Program 1: WAP to encrypt a message using Julis Ceaser Cipher.

```
1 alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
2 print("Lenght of alpha: {}".format(len(alpha)))
3
4 # input in capitalize
5 str_in = input("Enter a word, like HELLO:")
6 print("str_in = ", str_in)
7
8 msg_cipher = " "
9 n = len(str_in)
10 print("n =", n)
11
12 for i in range(n):
13     c = str_in[i]
14     loc = alpha.find(c)
15     print(i , c, loc) #This line can be omitted, used only to see
16     intermediate result
17     newloc = (loc+3)% 26
18     msg_cipher += alpha[newloc]
19
20 print("Plain text: ", str_in)
21 print("Cipher text: ", msg_cipher)
```

Listing 1: Ceaser Cipher.

Program 3

Program 3: WAP to encrypt a message using Ceaser Cipher with shift value.

```
1 alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
2 print("Lenght of alpha: {}".format(len(alpha)))
3
4 # input a word and it will change to uppercase
5 str_in = input("Enter a word, like HELLO:").upper()
6
7 # input a number in integer
8 shift = int(input("Input Shift value, like 3: "))
9
10 n = len(str_in)
11 msg_cipher = " "
12
13 for i in range(n):
14     c = str_in[i]
15     loc = alpha.find(c)
16     print(i , c, loc) # used to see intermediate result, can be omitted
17     newloc = (loc+shift) % 26
18     msg_cipher += alpha[newloc]
19
20 print("Plain text: ", str_in)
21 print("Shift Value: ", shift)
22 print("Cipher text: ", msg_cipher)
```

Listing 2: Ceaser Cipher with shift value.

Program 4

Program 4: WAP to encrypt and decrypt a message using Ceaser Cipher.

```
1 # Ceaser Cipher
2 alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
3
4 # define a functionn that checks if input message contains alhabets
  onlys
5 def contains_only_alphabets(input_str):
6     for char in input_str:
7         if not char.isalpha():
8             return False
9     return True
10
11 #input plaintext containing only alphabets
12 while True:
13     try:
14         msg = input("Input message: ")
15
16         #checking if input string contains alphabets only:dines
17         if contains_only_alphabets(msg):
18             break
19         else:
20             print("Input message does not contain alphabets only!!")
21     except ValueError:
22         print("Input is not an alphabetic. Try again!!!")
23
24 #input a postive integer (between 1 to 26) as key(shift)
25 while True:
26     try:
27         key = int(input("Enter a number between 1 and 26: "))
28         if 1 <= key <= 26:
```

```
29         break # Exit the loop if the input is valid
30     else:
31         print("Input is not between 1 and 26. Try again.")
32 except ValueError:
33     print("Input is not an integer. Try again.")
34
35 def ceasar_encryption(text, shift):
36     encrypted_text = ''
37
38     #changing text to upper (or lower) case
39     text = text.upper()
40
41     #length of text
42     n = len(text)
43
44     #encrypt text
45     for i in range(n):
46         c = text[i]
47         loc = alpha.find(c)
48         newloc = (loc + shift) % 26
49         encrypted_text += alpha[newloc]
50
51     return encrypted_text
52
53
54 def ceasar_decryption(encrypted_text, shift):
55     decrypted_text = ''
56
57
58     #length of encrypted_text
59     n = len(encrypted_text)
60
61     #decrypt text
62     for i in range(n):
63         c = encrypted_text[i]
64         loc = alpha.find(c)
65         newloc = (loc - shift) % 26
66         decrypted_text += alpha[newloc]
67
68     return decrypted_text
69
70 ciphertext = ceasar_encryption(msg, key)
71 decrypted_text = ceasar_decryption(ciphertext, key)
72 print("Plaintext: ", msg)
73 print("Ciphertext: ", ciphertext)
74 print("Decrypted text: ", decrypted_text)
```

Program 5

Program 5: WAP to perform brute force of a Ceaser cipher.

```
1 #caesar brute force
2
3 alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
4
5 def ceasar_decryption(encrypted_text, shift):
6     decrypted_text = ''
```

```
7
8
9     #length of encrypted_text
10    n = len(encrypted_text)
11
12    #decrypt text
13    for i in range(n):
14        c = encrypted_text[i]
15        loc = alpha.find(c)
16        newloc = (loc - shift) % 26
17        decrypted_text += alpha[newloc]
18
19    return decrypted_text
20
21 def caesar_brute_force(ciphertext):
22     for shift in range(26):
23         decrypted_text = caesar_decryption(ciphertext, shift)
24         print(f"Shift {shift}: {decrypted_text}")
25
26 #Example usage
27 ciphertext = 'ZRUOG'
28 caesar_brute_force(ciphertext)
```

Program 6

Program 6: WAP to perform encryption and decryption using Ceaser cipher.

```
1 def caesar_encrypt(text, shift):
2     encrypted_text = ""
3     for char in text:
4         if char.isalpha():
5             is_upper = char.isupper()
6             char = char.lower()
7             char_code = ord(char) #return unicode of character
8             encrypted_char = chr(((char_code - 97 + shift) % 26) + 97)
9             if is_upper:
10                 encrypted_char = encrypted_char.upper()
11             encrypted_text += encrypted_char
12         else:
13             encrypted_text += char
14     return encrypted_text
15
16 def caesar_decrypt(encrypted_text, shift):
17     decrypted_text = ""
18     for char in encrypted_text:
19         if char.isalpha():
20             is_upper = char.isupper()
21             char = char.lower()
22             char_code = ord(char)
23             decrypted_char = chr(((char_code - 97 - shift) % 26) + 97)
24             if is_upper:
25                 decrypted_char = decrypted_char.upper()
26             decrypted_text += decrypted_char
27         else:
28             decrypted_text += char
29     return decrypted_text
30
```

```
31 # Example usage:
32 plaintext = input("Input a message: ")
33 shift = 3 #change this value (1-26)
34 encrypted_text = caesar_encrypt(plaintext, shift)
35 decrypted_text = caesar_decrypt(encrypted_text, shift)
36
37 print("Original Text:", plaintext)
38 print("Cipher Text:", encrypted_text)
39 print("Decrypted Text:", decrypted_text)
```

Program 7

Program 7: WAP to perform encryption and decryption using Monoalphabetic cipher.

```
1 import random
2
3 # Define the alphabet
4 alphabet = "abcdefghijklmnopqrstuvwxyz"
5
6 # Create a random permutation of the alphabet
7 random.seed(42) # You can change the seed for a different permutation
8 cipher_alphabet = list(alphabet)
9 random.shuffle(cipher_alphabet)
10 cipher_alphabet = ''.join(cipher_alphabet)
11
12 print("Alphabet: ", alphabet)
13 print("Shuffled alphabet:", cipher_alphabet)
14
15 # Create a dictionary for the encryption and decryption mappings
16 encryption_mapping = {}
17 decryption_mapping = {}
18
19 for i in range(len(alphabet)):
20     encryption_mapping[alphabet[i]] = cipher_alphabet[i]
21     decryption_mapping[cipher_alphabet[i]] = alphabet[i]
22
23 print("Encryption mapping:\n ", encryption_mapping)
24 print("Decryption mapping: \n", decryption_mapping)
25
26 # Encrypt a message using the monoalphabetic cipher
27
28
29 def encrypt(message):
30     encrypted_message = ""
31
32     for char in message:
33         if char.isalpha():
34             # Preserve the case of the character
35             if char.isupper():
36                 encrypted_message += encryption_mapping[char.lower()].upper()
37             else:
38                 encrypted_message += encryption_mapping[char]
39         else:
40             encrypted_message += char
41
42     return encrypted_message
```

```
43
44
45 # Decrypt a message using the monoalphabetic cipher
46 def decrypt(ciphertext):
47     decrypted_message = ""
48
49     for char in ciphertext:
50         if char.isalpha():
51             # Preserve the case of the character
52             if char.isupper():
53                 decrypted_message += decryption_mapping[char.lower()].
upper()
54             else:
55                 decrypted_message += decryption_mapping[char]
56         else:
57             decrypted_message += char
58
59     return decrypted_message
60
61
62 # read message to encrypt:
63 message = input("Input message to encrypt: ")
64
65 # Encryption
66 cipher_text = encrypt(message)
67
68 # Decryption
69 decrypted_message = decrypt(cipher_text)
70
71 print("Plaintext: ", message)
72 print("Cipher text: ", cipher_text)
73 print("Decrypted text: ", decrypted_message)
```

Program 8

Program 8: WAP to perform encryption and decryption Playfair method.

```
1 def create_playfair_matrix(key):
2     key = key.replace(" ", "").upper()
3     alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ" # The letter 'J' is omitted
4
5     # Initialize the Playfair matrix with zeros
6     matrix = [['' for _ in range(5)] for _ in range(5)]
7
8     # Fill the matrix with the key
9     used_letters = set()
10    row, col = 0, 0
11
12    for letter in key:
13        if letter not in used_letters:
14            matrix[row][col] = letter
15            used_letters.add(letter)
16            col += 1
17            if col == 5:
18                col = 0
19                row += 1
20
```



```
21 # Fill the remaining empty cells with the alphabet
22 for letter in alphabet:
23     if letter not in used_letters:
24         matrix[row][col] = letter
25         used_letters.add(letter)
26         col += 1
27         if col == 5:
28             col = 0
29             row += 1
30
31 return matrix
32
33
34 def prepare_text(text):
35     text = text.replace(" ", "").upper()
36     # Replace 'J' with 'I'
37     text = text.replace("J", "I")
38     # Break the text into digrams
39     digrams = []
40     i = 0
41     while i < len(text):
42         if i + 1 < len(text) and text[i] == text[i + 1]:
43             digrams.append(text[i] + 'X')
44             i += 1
45         else:
46             digrams.append(text[i] + text[i + 1])
47             i += 2
48     return digrams
49
50
51 def encrypt_playfair(plaintext, key):
52     matrix = create_playfair_matrix(key)
53     plaintext = prepare_text(plaintext)
54     ciphertext = []
55
56     for digram in plaintext:
57         row1, col1 = None, None
58         row2, col2 = None, None
59
60         # Find the positions of the two letters in the matrix
61         for row in range(5):
62             for col in range(5):
63                 if matrix[row][col] == digram[0]:
64                     row1, col1 = row, col
65                 if matrix[row][col] == digram[1]:
66                     row2, col2 = row, col
67
68         # Handle the same row or column case
69         if row1 == row2:
70             col1 = (col1 + 1) % 5
71             col2 = (col2 + 1) % 5
72         elif col1 == col2:
73             row1 = (row1 + 1) % 5
74             row2 = (row2 + 1) % 5
75         else:
76             col1, col2 = col2, col1 # Swap columns
77
78         ciphertext.append(matrix[row1][col1] + matrix[row2][col2])
```

```
79
80     return ''.join(ciphertext)
81
82
83 def decrypt_playfair(ciphertext, key):
84     matrix = create_playfair_matrix(key)
85     ciphertext = prepare_text(ciphertext)
86     plaintext = []
87
88     for digram in ciphertext:
89         row1, col1 = None, None
90         row2, col2 = None, None
91
92         # Find the positions of the two letters in the matrix
93         for row in range(5):
94             for col in range(5):
95                 if matrix[row][col] == digram[0]:
96                     row1, col1 = row, col
97                 if matrix[row][col] == digram[1]:
98                     row2, col2 = row, col
99
100        # Handle the same row or column case
101        if row1 == row2:
102            col1 = (col1 - 1) % 5
103            col2 = (col2 - 1) % 5
104        elif col1 == col2:
105            row1 = (row1 - 1) % 5
106            row2 = (row2 - 1) % 5
107        else:
108            col1, col2 = col2, col1 # Swap columns
109
110        plaintext.append(matrix[row1][col1] + matrix[row2][col2])
111
112    return ''.join(plaintext)
113
114
115 # Example usage
116 key = "KEYWORD"
117 plaintext = "HELLO WORLD"
118 ciphertext = encrypt_playfair(plaintext, key)
119 decrypted_text = decrypt_playfair(ciphertext, key)
120
121 print("Original Text:", plaintext)
122 print("Encrypted Text:", ciphertext)
123 print("Decrypted Text:", decrypted_text)
```

LAB-2

FAMILIARIZATION HILL CIPHER

Objectives:

1. To be familiar with matrix operations, multiplication determinant, and inverse.
2. To encrypt a text using Hill cipher.

Program 9

Program 9: WAP to find determinant and inverse of a matrix using python library.

```
1 import numpy as np
2
3 # Define your matrix as a 2D NumPy array
4 # matrix = np.array([[2, 3, 1],
5 #                    [4, 1, 7],
6 #                    [0, 5, 2]])
7
8
9 matrix = np.array([[17, 17, 5],
10                  [21, 18, 21],
11                  [2, 2, 19]])
12
13
14 def determinant_matrix(matrix):
15     # Calculate the determinant
16     determinant = np.linalg.det(matrix)
17     return determinant
18
19
20 def inverse_matrix(matrix):
21     # Use the np.linalg.inv() function to find the inverse
22     inverse_matrix = np.linalg.inv(matrix)
23     return inverse_matrix
24
25
26 print("Determinant of a matrix:\n", determinant_matrix(matrix))
27 print("Inverse of a matrix:\n", inverse_matrix(matrix))
```

Program 10

Program 10: WAP to find determinant of a matrix.

```
1 '''
2 # 3 by 3 matrix
3 | a  b  c |
4 | d  e  f |
5 | g  h  i |
6
7 determinant
8
9 det = a(ei - fh) - b(di - fg) + c(dh - eg)
10
11 '''
12 import numpy as np
```

```
13
14 # Define your matrix as a 2D NumPy array
15 # matrix = np.array([[2, 3, 1],
16 #                    [4, 1, 7],
17 #                    [0, 5, 2]])
18
19 matrix = np.array([[6, 24, 1],
20                  [13, 16, 10],
21                  [20, 17, 15]])
22
23
24 def determinant_matrix(matrix):
25     # Calculate the determinant
26     # Calculate the determinant using the formula
27     det = ((matrix[0][0] * (matrix[1][1] * matrix[2][2] - matrix[1][2]
28 * matrix[2][1])) -
29           (matrix[0][1] * (matrix[1][0] * matrix[2][2] - matrix[1][2]
30 * matrix[2][0])) +
31           (matrix[0][2] * (matrix[1][0] * matrix[2][1] - matrix[1][1]
32 * matrix[2][0])))
33
34     return det
35
36 def inverse_matrix(matrix):
37     # Use the np.linalg.inv() function to find the inverse
38     inverse_matrix = np.linalg.inv(matrix)
39     return inverse_matrix
40
41 # Print the determinant
42 print("Determinant of a matrix:\n", determinant_matrix(matrix))
43 print("Inverse of a matrix:\n", inverse_matrix(matrix))
44
45 a = np. dot(determinant_matrix(matrix), matrix)
46
47 inv = inverse_matrix(a)
48 inv = (inv * determinant_matrix(matrix)) % 26
49 print(inv)
```

Program 11

Program 11: WAP to find inverse of a 2 by 2 matrix.

```
1 '''
2 # 2 by 2 matrix
3
4 # invers is given
5 1 / (ad - bc) * | d  -b |
6                 | -c  a  |
7
8
9 '''
10
11 import numpy as np
12
13 # Define your 2x2 matrix as nested lists
```

```
14 matrix = [[5, 10],
15           [9, 7]]
16
17 a = matrix[0][0]
18 b = matrix[0][1]
19 c = matrix[1][0]
20 d = matrix[1][1]
21
22 # Calculate the determinant
23
24 det = a * d - b * c
25
26
27 print("Matrix is:\n")
28 for row in matrix:
29     print(row)
30
31 print("Determinant is: ", det)
32
33
34 # Check if the determinant is zero (matrix is singular)
35 if det == 0:
36     print("The matrix is singular, and its inverse does not exist.")
37 else:
38     # Calculate the inverse
39     inverse = [[d / det, -b / det],
40               [-c / det, a / det]]
41
42     # Print the inverse matrix
43     print("Inverse Matrix:")
44     for row in inverse:
45         print(row)
```

Program 12

Program 12: WAP to find inverse of a 3 by 3 matrix.

```
1 '''
2 For a 3 by 3 matrix:
3 | a  b  c |
4 | d  e  f |
5 | g  h  i |
6
7
8 The inverse is given by:
9 1 / ((a * e * i) + (b * f * g) + (c * d * h) - (a * f * h) - (b * d * i
   ) - (c * e * g)) * | (e * i - f * h)  (c * h - b * i)  (b * f - c *
   e) |
10
11                       | (f * g - d * i)  (a * i - c * g)  (c * d - a * f) |
12                       | (d * h - e * g)  (b * g - a * h)  (a * e - b * d) |
13
14 '''
15
16 import numpy as np
```

```
17
18 # Set the global configuration for float display
19 np.set_printoptions(precision=4)
20
21 # matrix = np.array([[2, 3, 1],
22 #                    [4, 1, 7],
23 #                    [0, 5, 2]])
24
25 matrix = np.array([[17, 17, 5],
26                   [21, 18, 21],
27                   [2, 2, 19]])
28
29 a = matrix[0][0]
30 b = matrix[0][1]
31 c = matrix[0][2]
32
33 d = matrix[1][0]
34 e = matrix[1][1]
35 f = matrix[1][2]
36
37 g = matrix[2][0]
38 h = matrix[2][1]
39 i = matrix[2][2]
40
41
42 # Calculate the determinant
43 det = (a * e * i + b * f * g + c * d * h) - (a * f * h + b * d * i + c
44        * e * g)
45
46 print("Matrix is:\n")
47 for row in matrix:
48     print(row)
49
50 print("Determinant: ", det)
51
52 # Check if the determinant is zero (matrix is singular)
53 if det == 0:
54     print("The matrix is singular, and its inverse does not exist.")
55 else:
56     # Calculate the elements of the inverse matrix
57     inv_a = (e * i - f * h) / det
58     inv_b = (c * h - b * i) / det
59     inv_c = (b * f - c * e) / det
60     inv_d = (f * g - d * i) / det
61     inv_e = (a * i - c * g) / det
62     inv_f = (c * d - a * f) / det
63     inv_g = (d * h - e * g) / det
64     inv_h = (b * g - a * h) / det
65     inv_i = (a * e - b * d) / det
66
67     # Create the inverse matrix
68     inverse = [[inv_a, inv_b, inv_c],
69               [inv_d, inv_e, inv_f],
70               [inv_g, inv_h, inv_i]]
71
72 inverse_mod_26 = (inverse) % 26
73
```

```

74     # Print the inverse matrix
75     # print("Inverse Matrix:")
76     # for row in inverse:
77     #     print(row)
78
79     for row in inverse_mod_26:
80         print(row)

```

Program 13

Program 13: WAP to find inverse of a 3 by 3 matrix.

```

1  '''
2  For a 3 by 3 matrix:
3  | a  b  c |
4  | d  e  f |
5  | g  h  i |
6
7
8  The inverse is given by:
9  1 / ((a * e * i) + (b * f * g) + (c * d * h) - (a * f * h) - (b * d * i
   ) - (c * e * g)) * | (e * i - f * h)  (c * h - b * i)  (b * f - c *
   e) |
10
11                      | (f * g - d * i)  (a * i - c * g)  (c * d - a * f) |
12                      | (d * h - e * g)  (b * g - a * h)  (a * e - b * d) |
13
14  '''
15
16  import numpy as np
17
18  # Set the global configuration for float display
19  np.set_printoptions(precision=4)
20
21  # matrix = np.array([[2, 3, 1],
22  #                    [4, 1, 7],
23  #                    [0, 5, 2]])
24
25  matrix = np.array([[17, 17, 5],
26                    [21, 18, 21],
27                    [2, 2, 19]])
28
29  a = matrix[0][0]
30  b = matrix[0][1]
31  c = matrix[0][2]
32
33  d = matrix[1][0]
34  e = matrix[1][1]
35  f = matrix[1][2]
36
37  g = matrix[2][0]
38  h = matrix[2][1]
39  i = matrix[2][2]
40
41

```

```
42 # Calculate the determinant
43 det = (a * e * i + b * f * g + c * d * h) - (a * f * h + b * d * i + c
    * e * g)
44
45 print("Matrix is:\n")
46 for row in matrix:
47     print(row)
48
49 print("Determinant: ", det)
50
51 # Check if the determinant is zero (matrix is singular)
52 if det == 0:
53     print("The matrix is singular, and its inverse does not exist.")
54 else:
55     # Calculate the elements of the inverse matrix
56     inv_a = (e * i - f * h) / det
57     inv_b = (c * h - b * i) / det
58     inv_c = (b * f - c * e) / det
59     inv_d = (f * g - d * i) / det
60     inv_e = (a * i - c * g) / det
61     inv_f = (c * d - a * f) / det
62     inv_g = (d * h - e * g) / det
63     inv_h = (b * g - a * h) / det
64     inv_i = (a * e - b * d) / det
65
66     # Create the inverse matrix
67     inverse = [[inv_a, inv_b, inv_c],
68                [inv_d, inv_e, inv_f],
69                [inv_g, inv_h, inv_i]]
70
71
72 inverse_mod_26 = (inverse) % 26
73
74     # Print the inverse matrix
75     # print("Inverse Matrix:")
76     # for row in inverse:
77     #     print(row)
78
79     for row in inverse_mod_26:
80         print(row)
```

Program 14

Program 14: WAP to perform matrix multiplication using library function.

```
1 import numpy as np
2 # Set the desired number of decimal places globally for matrix value
  display
3 np.set_printoptions(precision=2)
4
5 # Define two matrices as NumPy arrays
6 matrix1 = np.array([[1, 2, 3],
7                      [4, 5, 6]])
8
9 matrix2 = np.array([[7, 8],
10                     [9, 10],
11                     [11, 12]])
```



```
12
13 # Perform matrix multiplication
14 result = np.dot(matrix1, matrix2)
15
16 # Print the result
17 print("Result of Matrix Multiplication:")
18 print(result)
```

Program 15

Program 15: WAP to perform matrix multiplication using library function.

```
1 import numpy as np
2
3 # Set the desired number of decimal places globally for matrix value
  display
4 np.set_printoptions(precision=2)
5
6 # Define two matrices as 2D lists
7 matrix1 = [[1, 2, 3],
8             [4, 5, 6]]
9
10 matrix2 = [[7, 8],
11            [9, 10],
12            [11, 12]]
13
14 # Determine the dimensions of the matrices
15 rows1 = len(matrix1)
16 cols1 = len(matrix1[0])
17 rows2 = len(matrix2)
18 cols2 = len(matrix2[0])
19
20 # Check if matrix multiplication is possible
21 if cols1 != rows2:
22     print("Matrix multiplication is not possible. Number of columns in
      matrix1 must be equal to the number of rows in matrix2.")
23 else:
24     # Create an empty result matrix
25     result = [[0 for _ in range(cols2)] for _ in range(rows1)]
26
27     # Perform matrix multiplication using nested loops
28     for i in range(rows1):
29         for j in range(cols2):
30             for k in range(cols1):
31                 result[i][j] += matrix1[i][k] * matrix2[k][j]
32
33     # Print the result
34     print("Result of Matrix Multiplication:")
35     for row in result:
36         print(row)
```

Program 16

Program 16: WAP to perform matrix multiplication using library function.

```
1 import numpy as np
2
```

```
3 # Define a function to convert a string to a list of numbers
4
5
6 def text_to_numbers(text):
7     text = text.lower().replace(" ", "")
8     text_to_num = []
9     for c in text:
10         loc = alpha.find(c)
11         # print(loc)
12         text_to_num.append(loc)
13     return text_to_num
14
15 # Define a function to convert a list of numbers to a string
16
17
18 def numbers_to_text(numbers):
19     return ''.join([chr(num + ord('a')) for num in numbers])
20
21 # Define a function for Hill Cipher encryption
22
23
24 def hill_cipher_encrypt(plaintext, key_matrix):
25     plaintext = plaintext.lower().replace(" ", "")
26     while len(plaintext) % 3 != 0:
27         plaintext += 'X' # Padding with 'X' if necessary
28
29     plaintext_numbers = text_to_numbers(plaintext)
30     print("Plain text in numbers: ", plaintext_numbers)
31     encrypted_numbers = []
32
33     for i in range(0, len(plaintext_numbers), 3):
34         block = np.array(plaintext_numbers[i:i+3])
35         encrypted_block = np.dot(block, key_matrix) % 26
36         print(encrypted_block)
37         encrypted_numbers.extend(encrypted_block)
38     print(encrypted_numbers)
39
40     return numbers_to_text(encrypted_numbers)
41
42
43 # Define the Hill Cipher key matrix (3x3 matrix)
44 key_matrix = np.array([[17, 17, 5],
45                        [21, 18, 21],
46                        [2, 2, 19]])
47
48 plaintext = "paymoremoney"
49 alpha = 'abcdefghijklmnopqrstuvwxyz'
50
51 text_to_num = text_to_numbers(plaintext)
52 print("Indexing of letters in message:\n")
53 print(text_to_num)
54
55 encrypted_text = hill_cipher_encrypt(plaintext, key_matrix)
56 print("Plain text: ", plaintext)
57 print("Encrypted:", encrypted_text.upper())
```

LAB-3

FAMILIARIZATION POLYALPHABETIC CIPHER

Objectives:

1. To perform encryption and decryption using Vigenere cipher and One-Time-Pad method
2. To encrypt and decrypt using Vernam method.

Program 17

Program 17: WAP to implement Vigenere cipher.

```
1 def vigenere_encrypt(plaintext, keyword):
2     plaintext = plaintext.upper()
3     keyword = keyword.upper()
4     encrypted_text = []
5     key_length = len(keyword)
6
7     for i in range(len(plaintext)):
8         if plaintext[i].isalpha():
9             shift = ord(keyword[i % key_length]) - ord('A')
10            encrypted_char = chr(
11                ((ord(plaintext[i]) - ord('A') + shift) % 26) + ord('A')
12            ))
13            encrypted_text.append(encrypted_char)
14        else:
15            encrypted_text.append(plaintext[i])
16
17    return ''.join(encrypted_text)
18
19 def vigenere_decrypt(ciphertext, keyword):
20     ciphertext = ciphertext.upper()
21     keyword = keyword.upper()
22     decrypted_text = []
23     key_length = len(keyword)
24
25     for i in range(len(ciphertext)):
26         if ciphertext[i].isalpha():
27             shift = ord(keyword[i % key_length]) - \
28                 ord('A') # ord('A') return value 65
29             decrypted_char = chr(
30                 ((ord(ciphertext[i]) - ord('A') - shift + 26) % 26) +
31                 ord('A'))
32             decrypted_text.append(decrypted_char)
33         else:
34             decrypted_text.append(ciphertext[i])
35
36    return ''.join(decrypted_text)
37
38 # Example usage
39 plaintext = input("Input a message: ")
40 keyword = "KEY" # you can change the key
41 encrypted_text = vigenere_encrypt(plaintext, keyword)
42 decrypted_text = vigenere_decrypt(encrypted_text, keyword)
```

```
43
44 print("Plaintext:", plaintext)
45 print("Encrypted text:", encrypted_text)
46 print("Decrypted text:", decrypted_text)
```

Program 18

Program 18: WAP to implement One Time Pad cipher.

```
1 import random
2
3
4 def generate_one_time_pad(length):
5     """Generate a random one-time pad of the specified length."""
6     return [random.randint(0, 25) for _ in range(length)]
7
8
9 def otp_encrypt(text, one_time_pad):
10    """Encrypt plaintext using the one-time pad."""
11    if len(text) != len(one_time_pad):
12        raise ValueError(
13            "Plaintext and one-time pad must be of the same length")
14
15    encrypted_text = []
16    for i in range(len(text)):
17        char = text[i].upper()
18        if char.isalpha():
19            shift = one_time_pad[i]
20            encrypted_char = chr(
21                ((ord(char) - ord('A') + shift) % 26) + ord('A'))
22            encrypted_text.append(encrypted_char)
23        else:
24            encrypted_text.append(char)
25
26    return ''.join(encrypted_text)
27
28
29 def otp_decrypt(ciphertext, one_time_pad):
30    """Decrypt ciphertext using the one-time pad."""
31    if len(ciphertext) != len(one_time_pad):
32        raise ValueError(
33            "Ciphertext and one-time pad must be of the same length")
34
35    decrypted_text = []
36    for i in range(len(ciphertext)):
37        char = ciphertext[i].upper()
38        if char.isalpha():
39            shift = one_time_pad[i]
40            decrypted_char = chr(
41                ((ord(char) - ord('A') - shift + 26) % 26) + ord('A'))
42            decrypted_text.append(decrypted_char)
43        else:
44            decrypted_text.append(char)
45
46    return ''.join(decrypted_text)
47
48
```

```
49 # Example usage
50 plaintext = input("Input a message: ")
51 one_time_pad = generate_one_time_pad(len(plaintext))
52 encrypted_text = otp_encrypt(plaintext, one_time_pad)
53 decrypted_text = otp_decrypt(encrypted_text, one_time_pad)
54
55
56 print("Plaintext:", plaintext)
57 print("Encrypted text:", encrypted_text)
58 print("Decrypted text:", decrypted_text)
```

Program 19

Program 18: WAP to implement Vernam cipher.

```
1 import random
2
3
4 def generate_random_key(message_length):
5     # Generate a random key of 0s and 1s of the same length as the
    message
6     return [random.randint(0, 1) for _ in range(message_length)]
7
8
9 def encrypt(plaintext, key):
10    # Ensure the key and plaintext have the same length
11    if len(plaintext) != len(key):
12        raise ValueError("Key length must match the plaintext length")
13
14    # Perform bitwise XOR to encrypt the plaintext
15    ciphertext = [str(int(plaintext[i]) ^ key[i])
16                  for i in range(len(plaintext))]
17
18    # Convert the list of bits back to a string
19    return ''.join(ciphertext)
20
21
22 def decrypt(ciphertext, key):
23    # Ensure the key and ciphertext have the same length
24    if len(ciphertext) != len(key):
25        raise ValueError("Key length must match the ciphertext length")
26
27    # Perform bitwise XOR to decrypt the ciphertext
28    plaintext = [str(int(ciphertext[i]) ^ key[i])
29                 for i in range(len(ciphertext))]
30
31    # Convert the list of bits back to a string
32    return ''.join(plaintext)
33
34
35 # Example usage
36 message = "Hello World!"
37 message_bits = [int(bit) for bit in ''.join(format(ord(char), '08b')
38                                               for char in message)] #
39    Convert message to bits
40 key = generate_random_key(len(message_bits)) # Generate a random key
```

```
41 key_stream = ''.join(str(bit) for bit in key)
42
43 ciphertext = encrypt(message_bits, key)
44 decrypted_message_bits = decrypt(ciphertext, key)
45
46 # Convert the decrypted bits back to a string
47 decrypted_message = ''.join(chr(int(
48     decrypted_message_bits[i:i+8], 2)) for i in range(0, len(
49     decrypted_message_bits), 8))
50
51 print("Original Message:", message)
52 print("length of message: ", len(message))
53 print("Key: ", key_stream)
54 print("Cipher text (in bits):", ''.join(map(str, ciphertext)))
55 print("Length of key: ", len(key))
56 print("Length of encrypted message: ", len(ciphertext))
57 print("Decrypted text:", decrypted_message)
58
59 Description: ciphertext = [str(int(plaintext[i]) ^ key[i]) for i in
60     range(len(plaintext))]
61
62 This line is responsible for generating the ciphertext, which is the
63 result of encrypting the plaintext using the Vernam cipher. It uses
64 a list comprehension to process each bit of the plaintext and
65 corresponding bit of the key and then combines them to form the
66 ciphertext.
67
68 Here's a step-by-step explanation:
69
70 1. 'for i in range(len(plaintext))': This part sets up a loop that
71 iterates over the indices
72 (positions) of each bit in the 'plaintext'. 'len(plaintext)' returns
73 the length of the plaintext,
74 and 'range(len(plaintext))' creates a sequence of indices from 0 to the
75 length of
76 the plaintext minus one.
77
78 2. 'plaintext[i]': Inside the loop, 'plaintext[i]' accesses the 'i'-th
79 bit of the plaintext.
80 This bit is either 0 or 1, representing the binary value of the
81 character in the plaintext.
82
83 3. 'key[i]': Similarly, 'key[i]' accesses the 'i'-th bit of the key.
84 The key is generated randomly and contains 0s and 1s, just like the
85 plaintext.
86
87 4. 'int(plaintext[i]) ^ key[i]': This part performs a bitwise XOR (
88 exclusive OR) operation between the 'i'-th bit of the plaintext
89 and the 'i'-th bit of the key. The 'int(...)' conversion is used to
90 ensure that
91 both 'plaintext[i]' and 'key[i]' are treated as integers. The XOR
92 operation returns 1 if the
93 bits being compared are different and 0 if they are the same.
94
95 5. 'str(...)': After performing the XOR operation,
96 the result is converted back to a string using 'str(...)'.
```

```
84 This is done because the ciphertext is typically represented as a
    sequence of characters.
85
86 So, in summary, this line processes each bit of the plaintext and key,
87 XORs them together, and converts the result to a string,
88 effectively creating the ciphertext as a sequence of 0s and 1s.
89
90 '''
```