IS PRACTICAL EXAM

1. LAN network using packet tracer
2. Wireshark
3. Euclidian algorithm
4. Extended Euclidian
5. Vernam cipher
6. Ceaser cipher
7. Rail fence
8. Columnar transposition
9. Row transposition
10. Product cipher
11. Playfair
12. RSA encryption decryption
13. RSA digital signature
14. Diffie Hellman Key exchange algo
15. Man in the middle Diffie Hellman
16. Euclidean Algorithm:

def euclidean\_algorithm(a, b):

    if b == 0:

        return a

    else:

        return euclidean\_algorithm(b, a % b)

# Taking user input for two numbers

try:

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

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

    # Calculate GCD using Euclidean algorithm

    gcd = euclidean\_algorithm(max(num1, num2), min(num1, num2))

    print(f"The GCD of {num1} and {num2} is: {gcd}")

except ValueError:

    print("Please enter valid integers.")

1. Extended Euclidean Algorithm

def extended\_euclidean(a, b):

    if b == 0:

        return a, 1, 0

    gcd, x1, y1 = extended\_euclidean(b, a % b)

    x = y1

    y = x1 - (a // b) \* y1

    return gcd, x, y

# Taking user input for two numbers

try:

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

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

    # Calculate GCD using Extended Euclidean algorithm

    gcd, x, y = extended\_euclidean(max(num1, num2), min(num1, num2))

    print(f"The GCD of {num1} and {num2} is: {gcd}")

    print(f"x = {x}, y = {y} such that {num1}\*{x} + {num2}\*{y} = {gcd}")

except ValueError:

    print("Please enter valid integers.")

1. Vernam Cipher

def Vernam(Plain, Key, Flag):

    result = ""

    for i in range(len(Plain)):

        char = Plain[i]

        if Flag:

            result += chr((ord(char) - 97 + ord(Key[i]) - 97) % 26 + 97)

        else:

            result += chr((ord(char) - ord(Key[i]) + 26) % 26 + 97)

    return result

if \_\_name\_\_ == "\_\_main\_\_":

    Key = ''.join(input("Enter Key: ").lower().split())

    Plain = ''.join(input("Enter PlainText: ").lower().split())

    if len(Key) != len(Plain):

        print("Invalid Key/PlainText length!")

        exit(None)

    CipherText = Vernam(Plain, Key, True)

    print("CipherText: ", CipherText)

    print("PlainText: ", Vernam(CipherText, Key, False))

1. Caesar Cipher

def encrypt(text, key):

    encrypted = ""

    for char in text:

        if 'A' <= char <= 'Z':

            encrypted\_char = chr((ord(char) - ord('A') + key) % 26 + ord('A'))

        elif 'a' <= char <= 'z':

            encrypted\_char = chr((ord(char) - ord('a') + key) % 26 + ord('a'))

        else:

            encrypted\_char = char

        encrypted += encrypted\_char

    return encrypted

def decrypt(encrypted\_text, key):

    decrypted = ""

    for char in encrypted\_text:

        if 'A' <= char <= 'Z':

            decrypted\_char = chr((ord(char) - ord('A') - key) % 26 + ord('A'))

        elif 'a' <= char <= 'z':

            decrypted\_char = chr((ord(char) - ord('a') - key) % 26 + ord('a'))

        else:

            decrypted\_char = char

        decrypted += decrypted\_char

    return decrypted

if \_\_name\_\_ == "\_\_main\_\_":

    text = "Krish"

    key = 5

    encrypted\_text = encrypt(text, key)

    print("Input text:", text)

    print("Encrypted text:", encrypted\_text)

    decrypted\_text = decrypt(encrypted\_text, key)

    print("Decrypted text:", decrypted\_text)

from collections import Counter

english\_frequencies = {'e': 0.127, 't': 0.091, 'a': 0.082, 'o': 0.075, 'i': 0.070, 'n': 0.067, 's': 0.063, 'h': 0.061, 'r': 0.060, 'd': 0.043, 'l': 0.040, 'c': 0.028, 'u': 0.028, 'm': 0.024, 'w': 0.023, 'f': 0.022, 'g': 0.020, 'y': 0.019, 'p': 0.019, 'b': 0.015, 'v': 0.010, 'k': 0.008, 'j': 0.002, 'x': 0.001, 'q': 0.001, 'z': 0.001}

def decrypt\_caesar(ciphertext, shift):

    decrypted\_text = ''

    for char in ciphertext:

        if char.isalpha():

            is\_upper = char.isupper()

            char = char.lower()

            decrypted\_char = chr(((ord(char) - shift - 97) % 26) + 97)

            if is\_upper:

                decrypted\_char = decrypted\_char.upper()

            decrypted\_text += decrypted\_char

        else:

            decrypted\_text += char

    return decrypted\_text

def calculate\_letter\_frequency(text):

    cleaned\_text = ''.join(char.lower() for char in text if char.isalpha())

    letter\_count = Counter(cleaned\_text)

    total\_letters = len(cleaned\_text)

    frequencies = {char: count / total\_letters for char, count in letter\_count.items()}

    return frequencies

def decrypt\_caesar\_with\_frequencies(ciphertext, probable\_letters):

    ciphertext\_frequencies = calculate\_letter\_frequency(ciphertext)

    for probable\_letter in probable\_letters:

        best\_shift = 0

        best\_correlation = 0

        shift = (ord(probable\_letter) - ord('a')) % 26

        decrypted\_text = decrypt\_caesar(ciphertext, shift)

        correlation = sum(english\_frequencies.get(char, 0) \* ciphertext\_frequencies.get(char, 0) for char in decrypted\_text.lower())

        if correlation > best\_correlation:

            best\_correlation = correlation

            best\_shift = shift

        if (text == decrypted\_text):

            print(f"Cipher Text: {encrypted\_text}\nMost probable letter: {probable\_letter}\nKey: {best\_shift}\nDecrypted Text: {decrypted\_text}")

if \_\_name\_\_ == "\_\_main\_\_":

    probable\_letters = english\_frequencies.keys()

    decrypt\_caesar\_with\_frequencies(encrypted\_text, probable\_letters)

1. Rail Fence Cipher

def encryptRailFence(text, key):

    rail = [['\n' for i in range(len(text))] for j in range(key)]

    dir\_down = False

    row, col = 0, 0

    for i in range(len(text)):

        if (row == 0) or (row == key - 1):

            dir\_down = not dir\_down

        rail[row][col] = text[i]

        col += 1

        if dir\_down:

            row += 1

        else:

            row -= 1

    result = []

    for i in range(key):

        for j in range(len(text)):

            if rail[i][j] != '\n':

                result.append(rail[i][j])

    return "".join(result)

def decryptRailFence(cipher, key):

    rail = [['\n' for i in range(len(cipher))] for j in range(key)]

    dir\_down = None

    row, col = 0, 0

    for i in range(len(cipher)):

        if row == 0:

            dir\_down = True

        if row == key - 1:

            dir\_down = False

        rail[row][col] = '\*'

        col += 1

        if dir\_down:

            row += 1

        else:

            row -= 1

    index = 0

    for i in range(key):

        for j in range(len(cipher)):

            if (rail[i][j] == '\*') and (index < len(cipher)):

                rail[i][j] = cipher[index]

                index += 1

    result = []

    row, col = 0, 0

    for i in range(len(cipher)):

        if row == 0:

            dir\_down = True

        if row == key - 1:

            dir\_down = False

        if rail[row][col] != '\*':

            result.append(rail[row][col])

            col += 1

        if dir\_down:

            row += 1

        else:

            row -= 1

    return "".join(result)

if \_\_name\_\_ == "\_\_main\_\_":

  pt="krishvalecha"

  key=2

  encrypt=encryptRailFence(pt, key)

  decrypt=decryptRailFence(encrypt, key)

  print("encrypted: ",encrypt)

  print("decrypted: ",decrypt)

1. Columnar transposition

import math

key = "this"

def encryptMessage(msg):

    cipher = ""

    k\_indx = 0

    msg\_len = float(len(msg))

    msg\_lst = list(msg)

    key\_lst = sorted(list(key))

    col = len(key)

    row = int(math.ceil(msg\_len / col))

    fill\_null = int((row \* col) - msg\_len)

    msg\_lst.extend('\_' \* fill\_null)

    matrix = [msg\_lst[i: i + col] for i in range(0, len(msg\_lst), col)]

    for \_ in range(col):

        curr\_idx = key.index(key\_lst[k\_indx])

        cipher += ''.join([row[curr\_idx] for row in matrix])

        k\_indx += 1

    return cipher

def decryptMessage(cipher):

    msg = ""

    k\_indx = 0

    msg\_indx = 0

    msg\_len = float(len(cipher))

    msg\_lst = list(cipher)

    col = len(key)

    row = int(math.ceil(msg\_len / col))

    key\_lst = sorted(list(key))

    dec\_cipher = [[None] \* col for \_ in range(row)]

    for \_ in range(col):

        curr\_idx = key.index(key\_lst[k\_indx])

        for j in range(row):

            dec\_cipher[j][curr\_idx] = msg\_lst[msg\_indx]

            msg\_indx += 1

        k\_indx += 1

    try:

        msg = ''.join(sum(dec\_cipher, []))

    except TypeError:

        raise TypeError("This program cannot handle repeating words.")

    null\_count = msg.count('\_')

    if null\_count > 0:

        return msg[: -null\_count]

    return msg

# Driver Code

msg = "krishvalecha"

cipher = encryptMessage(msg)

print("Encrypted Message: {}".format(cipher))

print("Decrypted Message: {}".format(decryptMessage(cipher)))

1. ROW TRANSPOSITION

def encrypt\_message(plaintext, key):

    key\_order = sorted(range(len(key)), key=lambda k: key[k])

    num\_rows = len(plaintext) // len(key) + int(len(plaintext) % len(key) != 0)

    matrix = [[''] \* len(key) for \_ in range(num\_rows)]

    # Pad the plaintext with underscores if needed

    if len(plaintext) % len(key) != 0:

        plaintext += '\_' \* (len(key) - len(plaintext) % len(key))

    idx = 0

    for r in range(num\_rows):

        for c in range(len(key)):

            matrix[r][c] = plaintext[idx]

            idx += 1

    encrypted\_message = ''

    for col in key\_order:

        for row in range(num\_rows):

            encrypted\_message += matrix[row][col]

    return encrypted\_message

def decrypt\_message(ciphertext, key):

    key\_order = sorted(range(len(key)), key=lambda k: key[k])

    num\_rows = len(ciphertext) // len(key)

    last\_row\_chars = len(ciphertext) % len(key)

    if last\_row\_chars != 0:

        num\_rows += 1

    matrix = [[''] \* len(key) for \_ in range(num\_rows)]

    idx = 0

    for col in key\_order:

        num\_chars = num\_rows - 1 if col >= last\_row\_chars else num\_rows

        for row in range(num\_chars):

            matrix[row][col] = ciphertext[idx]

            idx += 1

    # Fill the rest of the matrix cells with '\_'

    for row in range(num\_rows):

        for col in range(len(key)):

            if matrix[row][col] == '':

                matrix[row][col] = '\_'

    decrypted\_message = ''

    for r in range(num\_rows):

        for c in range(len(key)):

            decrypted\_message += matrix[r][c]

    return decrypted\_message

# Get user input for plaintext and key

plaintext = input("Enter the plaintext: ")

encryption\_key = input("Enter the key (as string): ")

encryption\_key = ''.join(filter(str.isalpha, encryption\_key))

encrypted\_text = encrypt\_message(plaintext, encryption\_key)

print("Encrypted Message:", encrypted\_text)

decrypted\_text = decrypt\_message(encrypted\_text, encryption\_key)

print("Decrypted Message:", decrypted\_text)

1. PRODUCT CIPHER

def encrypt(text, key):

encrypted = ""

for char in text:

if 'A' <= char <= 'Z':

encrypted\_char = chr((ord(char) - ord('A') + key) % 26 + ord('A'))

elif 'a' <= char <= 'z':

encrypted\_char = chr((ord(char) - ord('a') + key) % 26 + ord('a'))

else:

encrypted\_char = char

encrypted += encrypted\_char

return encrypted

if \_\_name\_\_ == "\_\_main\_\_":

text = "Krishvalecha"

key = 3

encrypted\_text = encrypt(text, key)

print(encrypted\_text)

def encryptRailFence(text, key):

rail = [['\n' for i in range(len(text))] for j in range(key)]

dir\_down = False

row, col = 0, 0

for i in range(len(text)):

if (row == 0) or (row == key - 1):

dir\_down = not dir\_down

rail[row][col] = text[i]

col += 1

if dir\_down:

row += 1

else:

row -= 1

result = []

for i in range(key):

for j in range(len(text)):

if rail[i][j] != '\n':

result.append(rail[i][j])

return "".join(result)

key=3

enc= encryptRailFence(encrypted\_text,key)

print(enc)

1. PLAYFAIR CIPHER

def toLowerCase(text):

    return text.lower()

def removeSpaces(text):

    return ''.join(char for char in text if char != ' ')

def Diagraph(text):

    Diagraph = []

    group = 0

    for i in range(2, len(text), 2):

        Diagraph.append(text[group:i])

        group = i

    Diagraph.append(text[group:])

    return Diagraph

def FillerLetter(text):

    k = len(text)

    if k % 2 == 0:

        for i in range(0, k, 2):

            if text[i] == text[i + 1]:

                new\_word = text[0:i + 1] + 'x' + text[i + 1:]

                new\_word = FillerLetter(new\_word)

                break

            else:

                new\_word = text

    else:

        for i in range(0, k - 1, 2):

            if text[i] == text[i + 1]:

                new\_word = text[0:i + 1] + 'x' + text[i + 1:]

                new\_word = FillerLetter(new\_word)

                break

            else:

                new\_word = text

    return new\_word

def generateKeyTable(word, list1):

    key\_letters = []

    for i in word:

        if i not in key\_letters:

            key\_letters.append(i)

    compElements = []

    for i in key\_letters:

        if i not in compElements:

            compElements.append(i)

    for i in list1:

        if i not in compElements:

            compElements.append(i)

    matrix = []

    while compElements != []:

        matrix.append(compElements[:5])

        compElements = compElements[5:]

    return matrix

def search(mat, element):

    for i in range(5):

        for j in range(5):

            if mat[i][j] == element:

                return i, j

def encrypt\_RowRule(matr, e1r, e1c, e2r, e2c):

    char1 = matr[e1r][(e1c + 1) % 5] if e1c != 4 else matr[e1r][0]

    char2 = matr[e2r][(e2c + 1) % 5] if e2c != 4 else matr[e2r][0]

    return char1, char2

def encrypt\_ColumnRule(matr, e1r, e1c, e2r, e2c):

    char1 = matr[(e1r + 1) % 5][e1c] if e1r != 4 else matr[0][e1c]

    char2 = matr[(e2r + 1) % 5][e2c] if e2r != 4 else matr[0][e2c]

    return char1, char2

def encrypt\_RectangleRule(matr, e1r, e1c, e2r, e2c):

    char1 = matr[e1r][e2c]

    char2 = matr[e2r][e1c]

    return char1, char2

def encryptByPlayfairCipher(Matrix, plainList):

    CipherText = []

    for i in range(0, len(plainList)):

        ele1\_x, ele1\_y = search(Matrix, plainList[i][0])

        ele2\_x, ele2\_y = search(Matrix, plainList[i][1])

        if ele1\_x == ele2\_x:

            c1, c2 = encrypt\_RowRule(Matrix, ele1\_x, ele1\_y, ele2\_x, ele2\_y)

        elif ele1\_y == ele2\_y:

            c1, c2 = encrypt\_ColumnRule(Matrix, ele1\_x, ele1\_y, ele2\_x, ele2\_y)

        else:

            c1, c2 = encrypt\_RectangleRule(Matrix, ele1\_x, ele1\_y, ele2\_x, ele2\_y)

        CipherText.append(c1 + c2)

    return ''.join(CipherText)

text\_Plain = 'hidethegold'

text\_Plain = removeSpaces(toLowerCase(text\_Plain))

PlainTextList = Diagraph(FillerLetter(text\_Plain))

if len(PlainTextList[-1]) != 2:

    PlainTextList[-1] = PlainTextList[-1] + 'z'

key = "helloworld"

key = toLowerCase(key)

Matrix = generateKeyTable(key, ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'])

CipherList = encryptByPlayfairCipher(Matrix, PlainTextList)

CipherText = ''.join(CipherList)

print("CipherText:", CipherText)

1. RSA ENCRYPTION DECRYPTION

import math

def gcd(a, h):

  temp = 0

  while(1):

    temp = a % h

    if (temp == 0):

      return h

    a = h

    h = temp

p = 3

q = 7

n = p\*q

e = 2

phi = (p-1)\*(q-1)

while (e < phi):

  # e must be co-prime to phi and

  # smaller than phi.

  if(gcd(e, phi) == 1):

    break

  else:

    e = e+1

# Private key (d stands for decrypt)

# choosing d such that it satisfies

# d\*e = 1 + k \* totient

k = 2

d = (1 + (k\*phi))/e

# Message to be encrypted

msg = 12.0

print("Message data = ", msg)

# Encryption c = (msg ^ e) % n

c = pow(msg, e)

c = math.fmod(c, n)

print("Encrypted data = ", c)

m = pow(c, d)

m = math.fmod(m, n)

print("Original Message Sent = ", m)

1. DIFFIE HELLMAN
2. # Diffie-Hellman Code
3. def prime\_checker(p):
4. # Checks If the number entered is a Prime Number or not
5. if p < 1:
6. return -1
7. elif p > 1:
8. if p == 2:
9. return 1
10. for i in range(2, p):
11. if p % i == 0:
12. return -1
13. return 1
14. def primitive\_check(g, p, L):
15. # Checks If The Entered Number Is A Primitive Root Or Not
16. for i in range(1, p):
17. L.append(pow(g, i) % p)
18. for i in range(1, p):
19. if L.count(i) > 1:
20. L.clear()
21. return -1
22. return 1
23. l = []
24. while 1:
25. P = int(input("Enter P : "))
26. if prime\_checker(P) == -1:
27. print("Number Is Not Prime, Please Enter Again!")
28. continue
29. break
30. while 1:
31. G = int(input(f"Enter The Primitive Root Of {P} : "))
32. if primitive\_check(G, P, l) == -1:
33. print(f"Number Is Not A Primitive Root Of {P}, Please Try Again!")
34. continue
35. break
36. # Private Keys
37. x1, x2 = int(input("Enter The Private Key Of User 1 : ")), int(
38. input("Enter The Private Key Of User 2 : "))
39. while 1:
40. if x1 >= P or x2 >= P:
41. print(f"Private Key Of Both The Users Should Be Less Than {P}!")
42. continue
43. break
44. # Calculate Public Keys
45. y1, y2 = pow(G, x1) % P, pow(G, x2) % P
46. # Generate Secret Keys
47. k1, k2 = pow(y2, x1) % P, pow(y1, x2) % P
48. print(f"\nSecret Key For User 1 Is {k1}\nSecret Key For User 2 Is {k2}\n")
49. if k1 == k2:
50. print("Keys Have Been Exchanged Successfully")
51. else:
52. print("Keys Have Not Been Exchanged Successfully")

L. MAN IN THE MIDDLE ATTACK

def deffie(a,b,g,p):

  Xa=pow(g,a)%p

  Xb=pow(g,b)%p

  return Xa, Xb

p=23

g=9

priv\_a = 5

priv\_b = 6

priv\_c=11

priv\_d=13

Xa,Xb=deffie(priv\_a,priv\_b,g,p)

print("Alice sends Xa to Bob",Xa)

print("Bob sends Xb to Alice",Xb)

Xc,Xd=deffie(priv\_c,priv\_d,g,p)

print("Darth intercepts Xa and sends Xc to Bob",Xc)

print("Darth intercepts Xb and sends Xd to Alice",Xd)

DKa=pow(Xa,priv\_d)%p

DKb=pow(Xb,priv\_c)%p

print("Darth calculates secret key of ALice: ",DKa)

print("Darth calculates secret key of Bob: ",DKb)

Ka=pow(Xd,priv\_a)%p

Kb=pow(Xc,priv\_b)%p

print("Alice calculates secret key unaware of the attack: ",Ka)

print("Bob calculates secret key unaware of the attack: ",Kb)