A Code

A.1 Back-end Code

```python
# David Budnitsky
# 20453508

import numpy as np
import u20453508_Prac3_RC4 as RC4


# region helperFunctions
def circularRightShift(num, shifts, numBits=64):
    """
    Right circular right-bit shit
    :param numBits:
    :param num:
    :param shifts:
    :return:
    """
    return (num >> shifts) | (num << (numBits - shifts)) & (int(2 ** numBits) - 1)


def circularLeftShift(num, shifts, numBits=64):
    """
    Left circular bit shift
    :param numBits:
    :param num:
    :param shifts:
    :return:
    """
    return (num << shifts) | (num >> (numBits - shifts))


# endregion helperFunctions

# region sha
def sha_Preprocess_Message(inputHex: str) -> str:
    """
    Takes in a hex input and pads it according to the SHA standard
    :param inputHex: Input message as a hex-string, no padding
    :return: The hex-string of the padded input
    """
    messageLen = len(inputHex) * 4
    inputBin = bin(int(inputHex, 16))[2:].zfill(messageLen)

    k = (896 - messageLen - 1) % 1024

    padding = '1' + '0' * k + bin(messageLen)[2:].zfill(128)

    ans = inputBin + padding
    ansLen = len(ans) // 4

    ans = int(ans, 2)
    ans = hex(ans)[2:].zfill(ansLen)
```

```python
    return ans


def sha_Create_Message_Blocks(inputHex: str) -> np.ndarray:
    """
    Breaks the message into blocks of 1024 bits, 256 hits per block
    :param inputHex: Preprocessed inputHex hex string
    :return: np array of hex strings, each with 256 characters
    """
    ans = np.array([inputHex[k:k + 256] for k in range(0, len(inputHex), 256)])
    return ans


def sha_Message_Schedule(inputHex: str) -> np.ndarray:
    """
    Makes the message schedule for a block from the inputHex.
    The first 16 message schedule pieces use 64-bit (16 hit) pieces of the message block
    :param inputHex: Input hex value to make the 80 message words from. This should always have a length of 1024.
    :return: Array of 80 words
    """
    W = [inputHex[k:k + 16] for k in range(0, len(inputHex), 16)]
    for k in range(16, 80):
        thisW = [W[k - t] for t in (16, 15, 7, 2)]
        temp = int(thisW[0], 16) + int(thisW[2], 16)

        x = int(thisW[1], 16)
        x1 = circularRightShift(x, 1)
        x2 = circularRightShift(x, 8)
        x3 = x >> 7
        temp1 = (x1 ^ x2 ^ x3)

        x = int(thisW[3], 16)
        x1 = circularRightShift(x, 19)
        x2 = circularRightShift(x, 61)
        x3 = x >> 6
        temp2 = (x1 ^ x2 ^ x3)

        temp = temp + temp1 + temp2
        temp = temp % int(2 ** 64)
        temp = hex(temp)[2:].upper().zfill(16)

        W.append(temp)

    W = np.array(W)
    return W


def sha_Hash_Round_Function(messageWordHex: str, aHex: str, bHex: str, cHex: str, dHex: str, eHex: str, fHex: str,
                            gHex: str, hHex: str, roundConstantHex: str) -> tuple:
    """
    Performs the Hash round function for SHA. This is seen in figure 11.11 in the textbook.
```

```
    pdf page 361.
    :param messageWordHex: Self-explanatory
    :param aHex: Self-explanatory
    :param bHex: Self-explanatory
    :param cHex: Self-explanatory
    :param dHex: Self-explanatory
    :param eHex: Self-explanatory
    :param fHex: Self-explanatory
    :param gHex: Self-explanatory
    :param hHex: Self-explanatory
    :param roundConstantHex: Self-explanatory
    :return: Tuple of new a-h as hex strings, each of 64 bits, 16 hits
    """
    a = int(aHex, 16)
    b = int(bHex, 16)
    c = int(cHex, 16)
    d = int(dHex, 16)
    e = int(eHex, 16)
    f = int(fHex, 16)
    g = int(gHex, 16)
    h = int(hHex, 16)

    wt = int(messageWordHex, 16)
    kt = int(roundConstantHex, 16)
    ch = (e & f) ^ ((~e) & g)
    maj = (a & b) ^ (a & c) ^ (b & c)
    sigma0 = circularRightShift(a, 28) ^ circularRightShift(a, 34) ^ circularRightShift(a, 39)
    sigma1 = circularRightShift(e, 14) ^ circularRightShift(e, 18) ^ circularRightShift(e, 41)

    T1 = (h + ch + sigma1 + wt + kt) % int(2 ** 64)
    T2 = (sigma0 + maj) % int(2 ** 64)

    hNew = hex(g)[2:].upper().zfill(16)
    gNew = hex(f)[2:].upper().zfill(16)
    fNew = hex(e)[2:].upper().zfill(16)
    eNew = hex((d + T1) % int(2 ** 64))[2:].upper().zfill(16)
    dNew = hex(c)[2:].upper().zfill(16)
    cNew = hex(b)[2:].upper().zfill(16)
    bNew = hex(a)[2:].upper().zfill(16)
    aNew = hex((T1 + T2) % int(2 ** 64))[2:].upper().zfill(16)

    ans = (aNew, bNew, cNew, dNew, eNew, fNew, gNew, hNew)
    return ans


def sha_F_Function(messageBlock: str, aHex: str, bHex: str, cHex: str, dHex: str, eHex: str, fHex: str, gHex: str,
                   hHex: str) -> tuple:
    W = sha_Message_Schedule(messageBlock)

    # Get the round constants as well
    roundConstants = [
        '428a2f98d728ae22', '7137449123ef65cd', 'b5c0fbcfec4d3b2f', 'e9b5dba58189dbbc',
        '3956c25bf348b538', '59f111f1b605d019', '923f82a4af194f9b', '
```

```
ab1c5ed5da6d8118',
    'd807aa98a3030242', '12835b0145706fbe', '243185be4ee4b28c', '550c7dc3d5ffb4e2',
    '72be5d74f27b896f', '80deb1fe3b1696b1', '9bdc06a725c71235', 'c19bf174cf692694',
    'e49b69c19ef14ad2', 'efbe4786384f25e3', '0fc19dc68b8cd5b5', '240ca1cc77ac9c65',
    '2de92c6f592b0275', '4a7484aa6ea6e483', '5cb0a9dcbd41fbd4', '76f988da831153b5',
    '983e5152ee66dfab', 'a831c66d2db43210', 'b00327c898fb213f', 'bf597fc7beef0ee4',
    'c6e00bf33da88fc2', 'd5a79147930aa725', '06ca6351e003826f', '142929670a0e6e70',
    '27b70a8546d22ffc', '2e1b21385c26c926', '4d2c6dfc5ac42aed', '53380d139d95b3df',
    '650a73548baf63de', '766a0abb3c77b2a8', '81c2c92e47edaee6', '92722c851482353b',
    'a2bfe8a14cf10364', 'a81a664bbc423001', 'c24b8b70d0f89791', 'c76c51a30654be30',
    'd192e819d6ef5218', 'd69906245565a910', 'f40e35855771202a', '106aa07032bbd1b8',
    '19a4c116b8d2d0c8', '1e376c085141ab53', '2748774cdf8eeb99', '34b0bcb5e19b48a8',
    '391c0cb3c5c95a63', '4ed8aa4ae3418acb', '5b9cca4f7763e373', '682e6ff3d6b2b8a3',
    '748f82ee5defb2fc', '78a5636f43172f60', '84c87814a1f0ab72', '8cc702081a6439ec',
    '90befffa23631e28', 'a4506cebde82bde9', 'bef9a3f7b2c67915', 'c67178f2e372532b',
    'ca273eceea26619c', 'd186b8c721c0c207', 'eada7dd6cde0eb1e', 'f57d4f7fee6ed178',
    '06f067aa72176fba', '0a637dc5a2c898a6', '113f9804bef90dae', '1b710b35131c471b',
    '28db77f523047d84', '32caab7b40c72493', '3c9ebe0a15c9bebc', '431d67c49c100d4c',
    '4cc5d4becb3e42b6', '597f299cfc657e2a', '5fcb6fab3ad6faec', '6c44198c4a475817']
for k in range(0, 80):
    aHex, bHex, cHex, dHex, eHex, fHex, gHex, hHex = sha_Hash_Round_Function(W[k],

        aHex,

        bHex,

        cHex,

        dHex,

        eHex,

        fHex,

        gHex,

        hHex,

```

```
                roundConstants[k])

    ans = (aHex, bHex, cHex, dHex, eHex, fHex, gHex, hHex)
    return ans


def sha_Process_Message_Block(inputHex: str, aHex: str, bHex: str, cHex: str, dHex: str, eHex: str, fHex: str,
                              gHex: str, hHex: str) -> tuple:
    f"""
    Performs sha_F_Function() on the input block then adds a-h to the new a-h.

    :param inputHex: Message block
    :param aHex: Current value of a
    :param bHex: Current value of b
    :param cHex: Current value of c
    :param dHex: Current value of d
    :param eHex: Current value of e
    :param fHex: Current value of f
    :param gHex: Current value of g
    :param hHex: Current value of h
    :return: New a-h values
    """

    oldH = np.array([aHex, bHex, cHex, dHex, eHex, fHex, gHex, hHex])

    newH = sha_F_Function(inputHex, aHex, bHex, cHex, dHex, eHex, fHex, gHex, hHex)
    ans1 = [hex((int(oldH[i], 16) + int(newH[i], 16)) % int(2 ** 64))[2:].upper().zfill(16) for i in range(0, 8)]

    ans = tuple(ans1)
    return ans


def sha_Calculate_Hash(inputHex: str) -> str:
    """
    Calculates the hash of the hex string provided.
    Initialises
    aHex
    bHex
    cHex
    dHex
    eHex
    fHex
    gHex
    hHex
    and then finds the hash.

    You must:
    initialise a-h
    preprocess input
    create blocks
    find the hash, update a-h for each block

    :param inputHex: Input of any lenght
```

```
    :return:
    """

    a = "6A09E667F3BCC908"
    b = "BB67AE8584CAA73B"
    c = "3C6EF372FE94F82B"
    d = "A54FF53A5F1D36F1"
    e = "510E527FADE682D1"
    f = "9B05688C2B3E6C1F"
    g = "1F83D9ABFB41BD6B"
    h = "5BE0CD19137E2179"

    inputHex = sha_Preprocess_Message(inputHex)
    messageBlocks = sha_Create_Message_Blocks(inputHex)

    for messageBlock in messageBlocks:
        a, b, c, d, e, f, g, h = sha_Process_Message_Block(messageBlock, a, b, c, d, e, f, g, h)

    ans = a + b + c + d + e + f + g + h
    return ans


def sha_String_To_Hex(inputStr: str) -> str:
    ans = ""
    for char in inputStr:
        temp = hex(ord(char))[2:].upper().zfill(2)
        ans = ans + temp
    return ans


def sha_Image_To_Hex(inputImg: np.ndarray) -> str:
    inputImg = inputImg.flatten()
    ans = ""
    for k in inputImg:
        ans = ans + hex(k)[2:].upper().zfill(2)
    return ans


def sha_Hex_To_Str(inputHex: str) -> str:
    inputBlocks = [inputHex[k:k + 2] for k in range(0, len(inputHex), 2)]
    ans = ""
    for k in inputBlocks:
        k = chr(int(k, 16))
        ans += k
    return ans


def sha_Hex_To_Im(inputHex: str, originalShape: tuple) -> np.ndarray:
    if len(inputHex) % 2 == 1:
        inputHex = '0' + inputHex

    inputBytes = np.array([int(inputHex[i:i + 2], 16) for i in range(0, len(inputHex), 2)])

    inputBytes = inputBytes.reshape(originalShape).round(0).astype(dtype=int)
```

```
    return inputBytes
# endregion sha


# region Transmitter
class Transmitter:
    def __init__(self, ):
        return

    def encrypt_With_RSA(self, message: str, RSA_Key: tuple) -> np.ndarray
    :
        """
        Receives a string of hex characters and encrypts with RSA, 2 bytes
     at a time. Block sze is 2 bytes, 4 hex characters.

        :param message: Hex string to encrypt. No padding, message will be
     a multiple of 4 hex chars.
        :param RSA_Key: RSA public key, (e, n)
        :return: 1D int array with encrypted message blocks.
        """
        m_blocks = [int(message[k:k + 4], 16) for k in range(0, len(
    message), 4)]
        e, n = RSA_Key
        C = [int(i ** e) % n for i in m_blocks]

        C = np.array(C)
        C = C.round(0).astype(int)
        return C

    def create_Digest(self, message) -> str:
        pranks = "this is an easter egg"
        if type(message) == type(pranks):
            inputHex = sha_String_To_Hex(message)
        else:
            inputHex = sha_Image_To_Hex(message)

        temp = sha_Calculate_Hash(inputHex)
        digest = inputHex + temp
        return digest

    def encrypt_with_RC4(self, digest: str, key: str) -> np.ndarray:
        """
        Encrypts the digest with RC4. The key is provided for RC4
        :param digest: M||H
        :param key: RC4 key
        :return:
        """
        cipher = RC4.rc4_Encrypt_String(digest, key)
        return cipher
# endregion Transmitter


# region Receiver
class Receiver:
    def __init__(self, ):
        self.p = 0
        self.q = 0
```

```
        self.n = 0
        self.phi = 0
        self.e = 0
        self.d = 0
        self.publicKey = (0, 0)
        self.privateKey = (0, 0)

    def printRec(self):
        ans =   f"Entered p value: {self.p}\n" \
              + f"Entered q value: {self.q}\n" \
              + f"Calculated n value: {self.n}\n" \
              + f"Calculated phi value: {self.phi}\n" \
              + f"Calculated e value: {self.e}\n" \
              + f"Calculated d value: {self.d}\n" \
              + f"Calculated PU value: {self.publicKey}\n" \
              + f"Calculated PR value: {self.privateKey}\n"
        return ans

    def generate_RSA_Keys(self, newP: int, newQ: int):
        """
        Given the p and q values, find:
            p
            q
            n
            phi
            e
            d
            publicKey
            privateKey

        :param newP: new vbalue to go to p
        :param newQ: new value to go to q
        :return: nothing
        """
        n = newP * newQ
        phi = ((newP - 1) * (newQ - 1))

        if phi > (2 ** 16 - 1):
            e = 2 ** 16 - 1
        else:
            e = phi // 4 - 1

        while np.gcd(e, phi) != 1 and e < phi:
            e += 1

        if e >= phi:
            e = phi // 2 + 1
        while np.gcd(e, phi) != 1:
            e += 1

        d = pow(e, -1, phi)

        PU = (e, n)
        PR = (d, n)

        self.p = newP
        self.q = newQ
```

```
        self.n = n
        self.phi = phi
        self.e = e
        self.d = d
        self.publicKey = PU
        self.privateKey = PR

    def decrypt_With_RSA(self, message: np.ndarray, RSA_Key: tuple) -> str:
        """
        Will receive an array of ints that make up the ciphertext of the RC4 key.
        Will apply decryption to this key.
        P = C^d mod n
        Encryption is done 2 bytes at a time, so I assume that the same holds for decryption, hence the .zfill(4)
        :param message: Int array of values to decrypt
        :param RSA_Key: Private key for decryption
        :return: Hex string version of P
        """
        P = ""
        d, n = RSA_Key
        for block in message:
            p = int(int(block) ** d) % n
            p = hex(p)[2:].upper().zfill(4)
            P = P + p
        return P

    def decrypt_With_RC4(self, digest: np.ndarray, key: str) -> str:
        plaintext = RC4.rc4_Decrypt_String(digest, key)
        return plaintext

    def split_Digest(self, digest: str) -> tuple:
        M = digest[0:-128]
        H = digest[-128:]

        ans = (M, H)
        return ans

    def authenticate_Message(self, digest: str) -> tuple:
        M, H = self.split_Digest(digest)
        h_calculated = sha_Calculate_Hash(M)
        auth = (H == h_calculated)
        ans = (auth, M, H, h_calculated)
        return ans
# endregion Receiver
```

A.2 Simulator Code

```python
# David Budnitsky
# 20453508

from u20453508_Prac_3_Backend import *
import numpy as np

# These values are from https://stackoverflow.com/questions/287871/how-do-i-print-colored-text-to-the-terminal
HEADER = '\033[95m'
OKBLUE = '\033[94m'
OKCYAN = '\033[96m'
OKGREEN = '\033[92m'
WARNING = '\033[93m'
FAIL = '\033[91m'
ENDC = '\033[0m'
BOLD = '\033[1m'
UNDERLINE = '\033[4m'

np.set_printoptions(threshold=np.inf)


def isPrime(num: int) -> bool:
    if num == 1:
        return False
    if num == 2:
        return True
    for k in range(2, num // 2 + 1):
        if np.gcd(k, num) != 1:
            return False
    return True


# def transmitMessage(message: np.ndarray) -> np.ndarray:
#     number = np.random.randint(0, 10)
#     if number == 5:
#         errorBytes = message[0:4]
#         errorBytes = [bin(k)[2:].zfill(8) for k in errorBytes]
#         errorPos = np.random.randint()


receiver = Receiver()
transmitter = Transmitter()

print(f"{HEADER}Welcome to Dodgy Dave's Dubious Digital Deception.{ENDC}\nLet's get started!\n\n")
print("To initialise a secure transmission channel, please give us the following: ")
p_input = input(f"{OKBLUE}RECEIVER{ENDC} p value, a good choice is 23:")
q_input = input(f"{OKBLUE}RECEIVER{ENDC} q value, a good choice is 3449:")

if p_input:
    p = int(p_input)
else:
    p = 23

```

```python
if q_input:
    q = int(q_input)
else:
    q = 3449

if (not isPrime(p)) or (not isPrime(q)) or ((p * q) < int(2 ** 16 - 1)):
    print(f"{FAIL}A condition has been violated, setting p and q to 23 and 3449 respectively.{ENDC}")
    p = 23
    q = 3449

receiver.generate_RSA_Keys(p, q)

print(f"{OKCYAN}\n\nPHASE 1\n\n{ENDC}")

print(receiver.printRec())
publicKey = receiver.publicKey

RC4_K = input(f"{OKGREEN}TRANSMITTER{ENDC} Enter the RC4 Key: ")
if not (RC4_K):
    print("Nothing entered, setting key to random thing.")
    RC4_K = "qwertyuio"
if len(RC4_K) % 2 == 1:
    print("To encrpyt the key, it must have an even number of bytes, adding a pad to the key.")
    RC4_K = "0" + RC4_K

RC4_Khex = sha_String_To_Hex(RC4_K)
RC4_K_enc = transmitter.encrypt_With_RSA(RC4_Khex, publicKey)
RC4_K_dec = receiver.decrypt_With_RSA(RC4_K_enc, receiver.privateKey)

print(f"{OKGREEN}TRANSMITTER{ENDC} RC4 Key in hex: {RC4_Khex}")
print(f"{OKGREEN}TRANSMITTER{ENDC} RC4 Key (encrypted): {RC4_K_enc}")
print(f"{OKBLUE}RECEIVER{ENDC} RC4 Key (decrypted): {RC4_K_dec}")

print(f"{OKCYAN}\n\nPHASE 2\n\n{ENDC}")

M = input(f"{OKGREEN}TRANSMITTER{ENDC} Enter a message: ")
if (not M):
    print(f"{FAIL}\nYou should have entered a valid message!\n{ENDC}")
    M = "In cryptography, encryption is the process of encoding " \
        "information. This process converts the original representation of the information, known as plaintext, " \
        "into an alternative form known as ciphertext. Ideally, only authorized parties can decipher a ciphertext " \
        "back to plaintext and access the original information. Encryption does not itself prevent interference but " \
        "denies the intelligible content to a would-be interceptor. For technical reasons, an encryption scheme " \
        "usually uses a pseudo-random encryption key generated by an algorithm. It is possible to decrypt the message " \
        "without possessing the key but, for a well-designed encryption scheme, considerable computational resources " \
        "and skills are required. An authorized recipient can easily decrypt the message with the key provided by the " \
        "originator to recipients but not to unauthorized users. Historically, various forms of encryption have been " \
```

```
        "used to aid in cryptography. Early encryption techniques were often used in military messaging. Since then, " \
        "new techniques have emerged and become commonplace in all areas of modern computing.Modern encryption " \
        "schemes use the concepts of public-key and symmetric-key. Modern encryption techniques ensure security " \
        "because modern computers are inefficient at cracking the encryption. This text was taken from wikipedia."
print(f"{OKGREEN}TRANSMITTER: {ENDC}Message is \n{M}")
PM_hex = sha_String_To_Hex(M)
PM_hash = sha_Calculate_Hash(PM_hex)
P_Digest = PM_hex + PM_hash
C_digest = transmitter.encrypt_with_RC4(P_Digest, RC4_K)
print(f"{OKGREEN}TRANSMITTER: {ENDC}Plaintext message: \n{PM_hex}")
print(f"{OKGREEN}TRANSMITTER: {ENDC}Plaintext hash: \n{PM_hash}")
print(f"{OKGREEN}TRANSMITTER: {ENDC}Plaintext digest: \n{P_Digest}")
print(f"{OKGREEN}TRANSMITTER: {ENDC}Ciphertext digest: \n{C_digest}")

print(f"{OKCYAN}\n\nPHASE 3\n\n{ENDC}")

print(f"{OKBLUE}RECEIVER{ENDC} RC4 Key (decrypted): {RC4_K_dec}")
print(f"{OKBLUE}RECEIVER{ENDC} RC4 Key (decrypted): {RC4_K_dec}")
```

A.3 RC4 Code

```python
# David Budnitsky
# 20453508

import numpy as np


# region RC4
def rc4_Init_S_T(key: str) -> np.ndarray:
    """
    Generates initial S and T arrays. Returns a 2D array holding S and T
    in elements 0 and 1 respectively
    :param key: The encryption key
    :return: [S, T]
    """
    S = [i for i in range(0, 256)]

    K = [ord(k) for k in list(key)]

    T = np.array([])
    while len(T) < 256:
        T = np.concatenate((T, K))

    T = T[0:256]

    S = np.array(S).round(0).astype(int)
    T = np.array(T).round(0).astype(int)

    ans = np.array([S, T]).round(0).astype(int)
    return ans


def rc4_Init_Permute_S(sArray: np.ndarray, tArray: np.ndarray) -> np.
    ndarray:
    """
    Performs initial permutation on the S array
    :param sArray: S array
    :param tArray: T array
    :return: The permuted S array
    """

    j = 0
    for i in range(0, 256):
        j = (j + sArray[i] + tArray[i]) % 256
        temp = sArray[i]
        sArray[i] = sArray[j]
        sArray[j] = temp

    # sArray = np.array(sArray).round(0).astype(int)
    return np.asarray(sArray)


# returns (i, j, sArray, k)
def rc4_Generate_Stream_Iteration(i: int, j: int, sArray: np.ndarray) ->
    tuple:
    """
```

```
    Generates a random byte stream byte
    :param i: Value used in stream generation
    :param j: Value used in stream generation
    :param sArray: last Modified S array
    :return: tuple containing (i,j,sArray, k)
    """
    i = (i + 1) % 256
    j = (j + sArray[i]) % 256
    temp = sArray[i]
    sArray[i] = sArray[j]
    sArray[j] = temp

    t = (sArray[i] + sArray[j]) % 256
    k = sArray[t]

    return tuple((i, j, sArray, k))


def rc4_Process_Byte(byteToProcess: int, k: int) -> int:
    """
    :param byteToProcess: byte to be processed
    :param k: k value
    :return: biwise XOR of k and byteToProcess
    """
    return np.bitwise_xor(byteToProcess, k)


def rc4_Encrypt_String(plaintext: str, key: str) -> np.ndarray:
    """
    Example usage:
    P = "hello world"
    Phex = sha_String_To_Hex(P)
    C = rc4.rc4_Encrypt_String(Phex, "qwerty")
    Pdec = rc4.rc4_Decrypt_String(C, "qwerty")
    Pstr = sha_Hex_To_Str(Pdec)
    print(Pstr)
    :param plaintext: The plaintext to encrypt. Input is a hex string and encryption will be done byte by byte.
    :param key: The key to initalise S and T with
    :return: Encrypted text as an int np.ndarray
    """
    if len(plaintext) % 2 == 1:
        plaintext = '0' + plaintext

    P = [int(plaintext[i:i + 2], 16) for i in range(0, len(plaintext), 2)]

    S, T = rc4_Init_S_T(key)
    S = rc4_Init_Permute_S(S, T)

    C = []
    i = 0
    j = 0
    for byte in P:
        (i, j, S, k) = rc4_Generate_Stream_Iteration(i, j, S)
        c = rc4_Process_Byte(byte, k)
        C.append(c)
    C = np.array(C).round(0).astype(int)
```

```python
    return C


def rc4_Decrypt_String(ciphertext: np.ndarray, key: str) -> str:
    """
    Decrypts ciphertext using key provided.

    Example usage:
    P = "hello world"
    Phex = sha_String_To_Hex(P)
    C = rc4.rc4_Encrypt_String(Phex, "qwerty")
    Pdec = rc4.rc4_Decrypt_String(C, "qwerty")
    Pstr = sha_Hex_To_Str(Pdec)
    print(Pstr)

    :param ciphertext: Ciphertext to be decrypted, int np array
    :param key: Key to decrypt with
    :return: Hex-string plaintext.
    """
    S, T = rc4_Init_S_T(key)
    S = rc4_Init_Permute_S(S, T)

    i = 0
    j = 0
    P = ""
    for byte in ciphertext:
        (i, j, S, k) = rc4_Generate_Stream_Iteration(i, j, S)
        p = hex(rc4_Process_Byte(byte, k))[2:].upper().zfill(2)
        P = P + p
    return P


def rc4_Encrypt_Image(plaintext: np.ndarray, key: str) -> np.ndarray:
    """
    :param plaintext: 3D image array to encrypt
    :param key: Key to encrypt with
    :return: 1D array of ciphertext image
    """
    P = plaintext.flatten()
    S, T = rc4_Init_S_T(key)
    S = rc4_Init_Permute_S(S, T)

    C = []
    i = 0
    j = 0
    for char in P:
        (i, j, S, k) = rc4_Generate_Stream_Iteration(i, j, S)
        c = rc4_Process_Byte(char, k)
        C.append(c)
    C = np.array(C).round(0).astype(int)
    return C


def rc4_Decrypt_Image(ciphertext: np.ndarray, key: str) -> np.ndarray:
    """
    :param ciphertext: Ciphertext to decrypt as a 1D int array
    :param key: Key to use for decryption
```

```
    :return:
    """
    S, T = rc4_Init_S_T(key)
    S = rc4_Init_Permute_S(S, T)

    P = []
    i = 0
    j = 0
    for char in ciphertext:
        (i, j, S, k) = rc4_Generate_Stream_Iteration(i, j, S)
        c = rc4_Process_Byte(char, k)
        P.append(c)
    P = np.array(P).round(0).astype(int)
    return P

# endregion RC4
```