Vigenère cipher Attack

```
import pandas as pd

def find_repeats(text, min_length=3):
    repeats = {}
    for i in range(len(text) - min_length + 1):
        substr = text[i:i + min_length]
        if substr in repeats:
            repeats[substr].append(i)
        else:
            repeats[substr] = [i]
    return {substr: indices for substr, indices in repeats.items() if len(indices) > 1}

def find_distances(repeats):
    distances = {}
    for substr, indices in repeats.items():
        distances[substr] = [indices[j + 1] - indices[j] for j in range(len(indices) - 1)]
    return distances

def find_factors(distances):
    factors = {}
    for substr, dist_list in distances.items():
        factors[substr] = []
        for dist in dist_list:
            for i in range(2, dist + 1):
                if dist % i == 0 and i not in factors[substr]:
                    factors[substr].append(i)
    return factors

def kasiski_table(text):
    repeats = find_repeats(text)
    distances = find_distances(repeats)
    factors = find_factors(distances)
    return factors

text = "TTEUM GQNDV EOIOL EDIRE MQTGS DAFDR CDYOX IZGZP PTAAI TUCSI XFBXY SUNFE SQRHI SAFHR TQRVS VQNBE EEAQG IBHDV SNARI DANSL EXESX EDSNJ A

kasiski = kasiski_table(text)
df = pd.DataFrame(kasiski.items(), columns=["Substring", "Factors"])
df["Factors"] = df["Factors"].apply(lambda x: ', '.join(map(str, x)))
df
```

	Substring	Factors
0	DV	2, 3, 4, 6, 8, 12, 16, 24, 32, 48, 96
1	L E	2, 3, 6, 17, 34, 51, 102
2	ED	2, 3, 4, 6, 9, 12, 18, 27, 36, 54, 108
3	DA	2, 3, 4, 6, 7, 12, 14, 21, 28, 42, 84
4	ZP	2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, ...
5	P P	2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, ...
6	PT	2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, ...
7	PTA	2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, ...
8	TAA	2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, ...
9	AAI	2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, ...
10	AI	2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, ...
11	I T	2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, ...
12	TU	2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, ...
13	TUC	2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, ...
14	X E	2, 3, 6, 9, 18
15	HX	2, 3, 6, 7, 14, 21, 42

Next steps: View recommended plots

```
import pandas as pd
from collections import Counter

    # Remove white spaces and convert text to uppercase

text = "TTEUM GQNDV EOIOL EDIRE MQTGS DAFDR CDYOX IZGZP PTAAI TUCSI XFBXY SUNFE SQRHI SAFHR TQRVS VQNBE EEAQG IBHDV SNARI DANSL EXESX EDSNJ
text = text.replace(" ", "").upper()

    # Count the occurrences of each letter
letter_count = Counter(text)


    # Create a DataFrame to store the letter frequencies
df = pd.DataFrame(list(letter_count.items()), columns=['Letter', 'Frequency'])

    # # Add a column for frequency - 1
df['Frequency - 1'] = df['Frequency'] - 1

    # # Add a column for the product of Frequency and Frequency - 1
df['Product'] = df['Frequency'] * df['Frequency - 1']

df



    # return df

# # Calculate letter frequencies and average product
# letter_freq_table = letter_frequency(text)
# print("Letter Frequency Table:")
# letter_freq_table
```

	Letter	Frequency	Frequency - 1	Product
0	T	10	9	90
1	E	17	16	272
2	U	5	4	20
3	M	2	1	2
4	G	4	3	12
5	Q	6	5	30
6	N	7	6	42
7	D	10	9	90
8	V	4	3	12
9	O	6	5	30
10	I	10	9	90
11	L	2	1	2
12	R	9	8	72
13	S	12	11	132
14	A	12	11	132
15	F	5	4	20
16	C	3	2	6
17	Y	7	6	42
18	X	9	8	72
19	Z	3	2	6
20	P	5	4	20
21	B	4	3	12
22	H	5	4	20
23	J	1	0	0
24	W	1	0	0
25	K	1	0	0

Next steps: View recommended plots

```
# Calculate IC
total = df['Product'].sum()
print("Total = ",total)
print("IC = ", total/(160*159))
```

```
Total =  1226
IC =  0.04819182389937107
```

```
def split_cipher_text(cipher_text):
    cipher_text = cipher_text.replace(" ", "").upper()
    alphabets = [''] * 5

    for i, char in enumerate(cipher_text):
        alphabet_index = i % 5
        alphabets[alphabet_index] += char

    return alphabets

cipher_text = "TTEUMGQNDVEOIOL EDIREMQTGSDAFDR CDYOXIZGZPPTAAI TUCSIXFBXYSUNFE SQRHISAFHRTQRVS VQNBEEEAQGIBHDV SNARIDANSLEXESX EDSNJAWEXAODD
alphabets = split_cipher_text(cipher_text)

for i, alphabet in enumerate(alphabets):
    print(f"Alphabet {i+1}: {alphabet}")
```

```
Alphabet 1: TGEEMDCIPTXSSSTVEISDEEAOEYROPTBU
Alphabet 2: TQODQADZTUFUQAQQEBNAXDWDYEYXTUEF
Alphabet 3: ENIITFYGACBNRFRNAHANESEDPAOYACTI
Alphabet 4: UDORGDOZASXFHHVBQDRSSNXHKEEZARHN
Alphabet 5: MVLESRXPIIYEIRSEGVILXJAXSSTPIYXR
```

```
import pandas as pd

def populate_frequency_table(sequences):
    # Initialize an empty DataFrame to store frequencies of characters
    frequency_df = pd.DataFrame(columns=[chr(i) for i in range(ord('A'), ord('Z')+1)])

    # Iterate through each sequence
    index = 0
    for sequence in sequences:
        index += 1
        # Initialize a dictionary to store frequencies of characters for this sequence
        frequency_table = {chr(i): 0 for i in range(ord('A'), ord('Z')+1)}

        # Convert the sequence to uppercase to ensure consistency
        sequence = sequence.upper()

        # Iterate through the sequence and update the frequency table
        for char in sequence:
            if char.isalpha():  # Check if the character is a letter
                frequency_table[char] += 1

        # Convert the dictionary to a DataFrame and append it to the main DataFrame
        sequence_df = pd.DataFrame.from_dict(frequency_table, orient='index').T
        sequence_df.columns = [chr(i) for i in range(ord('A'), ord('Z')+1)]
        sequence_df.index = ["Aplhabet " + str(index)]
        frequency_df = frequency_df.append(sequence_df, ignore_index=False)

    return frequency_df

frequency_df = populate_frequency_table(alphabets)
frequency_df
```

```
<ipython-input-109-c81d958b38ba>:26: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future ve
  frequency_df = frequency_df.append(sequence_df, ignore_index=False)
<ipython-input-109-c81d958b38ba>:26: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future ve
  frequency_df = frequency_df.append(sequence_df, ignore_index=False)
<ipython-input-109-c81d958b38ba>:26: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future ve
  frequency_df = frequency_df.append(sequence_df, ignore_index=False)
<ipython-input-109-c81d958b38ba>:26: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future ve
  frequency_df = frequency_df.append(sequence_df, ignore_index=False)
<ipython-input-109-c81d958b38ba>:26: FutureWarning: The frame.append method is deprecated and will be removed from pandas in a future ve
  frequency_df = frequency_df.append(sequence_df, ignore_index=False)
```

	A	B	C	D	E	F	G	H	I	J	...	Q	R	S	T	U	V	W	X	Y	Z
Aplhabet 1	1	1	1	2	6	0	1	0	2	0	...	0	1	4	4	1	1	0	1	1	0
Aplhabet 2	3	1	0	4	3	2	0	0	0	0	...	5	0	0	3	3	0	1	2	2	1
Aplhabet 3	5	1	2	1	3	2	1	1	3	0	...	0	2	1	2	0	0	0	0	2	0
Aplhabet 4	2	1	0	3	2	1	1	4	0	0	...	1	3	3	0	1	1	0	2	0	2
Aplhabet 5	1	0	0	0	3	0	1	0	5	1	...	0	3	4	1	0	2	0	4	2	0

5 rows × 26 columns

```
frequency_df_as_str = frequency_df.apply(lambda row: ' '.join(map(str, row)), axis=1)
frequency_df_as_str
Aplhabet_freq = []
for i in range(1,len(alphabets)+1):
    Aplhabet_freq.append(' '.join(map(str, frequency_df.loc['Aplhabet '+str(i)])))

Aplhabet_freq
```

```
['1 1 1 2 6 0 1 0 2 0 0 0 1 0 2 2 0 1 4 4 1 1 0 1 1 0',
 '3 1 0 4 3 2 0 0 0 0 0 0 0 1 1 0 5 0 0 3 3 0 1 2 2 1',
 '5 1 2 1 3 2 1 1 3 0 0 0 0 4 1 1 0 2 1 2 0 0 0 0 2 0',
 '2 1 0 3 2 1 1 4 0 0 1 0 0 2 2 0 1 3 3 0 1 1 0 2 0 2',
 '1 0 0 0 3 0 1 0 5 1 0 2 1 0 0 2 0 3 4 1 0 2 0 4 2 0']
```

```
index = 0
for i in Aplhabet_freq:
  index+= 1
  converted_sequence = ''
  for char in i:
        if char.isdigit():
            num = int(char)
            if num == 2:
                converted_sequence += 'M'
            elif 3 <= num <= 4:
                converted_sequence += 'H'
            else:
                converted_sequence += 'L'
        else:
            converted_sequence += char
  print("Alphabet", index, converted_sequence)
```

```
     Alphabet 1 L L L M L L L L M L L L L L M M L L H H L L L L L L
     Alphabet 2 H L L H H M L L L L L L L L L L L L L H H L L M M L
     Alphabet 3 L L M L H M L L H L L L L H L L L M L M L L L L M L
     Alphabet 4 M L L H M L L H L L L L L M M L L H H L L L L M L M
     Alphabet 5 L L L L H L L L L L L M L L L M L H H L L M L H M L
```

```
def Find_IC(text):
    text = text.replace(" ", "").upper()

    # Count the occurrences of each letter
    letter_count = Counter(text)

    # Create a DataFrame to store the letter frequencies
    df = pd.DataFrame(list(letter_count.items()), columns=['Letter', 'Frequency'])

    # Add a column for frequency - 1
    df['Frequency - 1'] = df['Frequency'] - 1

    # Add a column for the product of Frequency and Frequency - 1
    df['Product'] = df['Frequency'] * df['Frequency - 1']

    total = df['Product'].sum()
    IC = total / (160 * 159)
    return IC
index = 0
for i in alphabets:
    index += 1
    print("Alphabet", index ,"IC -", Find_IC(i))
```

```
     Alphabet 1 IC - 0.0024371069182389936
     Alphabet 2 IC - 0.0024371069182389936
     Alphabet 3 IC - 0.0021226415094339622
     Alphabet 4 IC - 0.0016509433962264152
     Alphabet 5 IC - 0.0025157232704402514
```

```
import itertools

def vigenere_decrypt(ciphertext, key):
    decrypted_text = ''
    key_length = len(key)
    key_index = 0

    for char in ciphertext:
        if char.isalpha():
            # Determine the shift value based on the corresponding character in the key
            shift = ord(key[key_index % key_length]) - ord('A')

            # Decrypt the character using the shift value
            decrypted_char = chr(((ord(char) - ord('A') - shift) % 26) + ord('A'))

            decrypted_text += decrypted_char

            # Move to the next character in the key
            key_index += 1
        else:
            # Non-alphabetic characters remain unchanged
            decrypted_text += char

    return decrypted_text

# Example ciphertext
ciphertext = "TTEUMGQNDVEOIOLEDIREMQTGSDAFDRCDYOXIZGZPPTAAITUCSIXFBXYSUNFESQRHISAFHRTQRVSVQNBEEEAQGIBHDVSNARIDANSLEXESXEDSNJAWEXAODDHXEYPKSY
decrypted_text = vigenere_decrypt(ciphertext, 'AMAZE')
print(f"Key: {key}, Decrypted text: {decrypted_text}")
```

```
RECIPHERISAMETHODOFENCRYPTINGALPHABETICTEXTBYUSINGASERIESOFINTERWOVENCAESARCIPHERSBASEDONTHELETTERSOFAKEYWORDITEMPLOYSAFORMOFPOLYALPHABET
```

RSA Algorithm Decryption

```
# Function to decrypt RSA ciphertext with only public key
def decrypt_with_public_key(ciphertext, public_key):
    e, n = public_key
    plaintext = pow(ciphertext, e, n)
    return plaintext

# Function to convert decimal to binary
def decimal_to_binary(decimal):
    binary = bin(decimal)[2:]
    return binary

# Function to convert binary to string
def binary_to_string(binary):
    n = int(binary, 2)
    return n.to_bytes((n.bit_length() + 7) // 8, 'big').decode()

# Provided public key
public_key = (4253529586511730793292182592897102643, 28948022309329048855892746252171976958893825396437940984840526105365295661081)

# Provided ciphertext
ciphertext = 6179930535625431814846047153483566738402890213958997535651208455989582499855

# Decrypt ciphertext using public key
plaintext_decimal = decrypt_with_public_key(ciphertext, public_key)

# Convert decimal plaintext to binary
plaintext_binary = decimal_to_binary(plaintext_decimal)

# Convert binary plaintext to string
plaintext_string = binary_to_string(plaintext_binary)

print("Plaintext (Decimal):", plaintext_decimal)
print("Plaintext (Binary):", plaintext_binary)
print("Plaintext (String):", plaintext_string)
```

```
Plaintext (Decimal): 369604964536956849050713
Plaintext (Binary): 1001110010001000101001101000101010000110101010101010010010010010101010001011001
Plaintext (String): NDSECURITY
```

DES Algorithm Decryption

```
# PC1 table for key permutation
PC1 = [57, 49, 41, 33, 25, 17, 9,
       1, 58, 50, 42, 34, 26, 18,
       10, 2, 59, 51, 43, 35, 27,
       19, 11, 3, 60, 52, 44, 36,
       63, 55, 47, 39, 31, 23, 15,
       7, 62, 54, 46, 38, 30, 22,
       14, 6, 61, 53, 45, 37, 29,
       21, 13, 5, 28, 20, 12, 4]

# LSH table for left shift
LSH = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]

# PC2 table for key permutation
PC2 = [14, 17, 11, 24, 1, 5,
       3, 28, 15, 6, 21, 10,
       23, 19, 12, 4, 26, 8,
       16, 7, 27, 20, 13, 2,
       41, 52, 31, 37, 47, 55,
       30, 40, 51, 45, 33, 48,
       44, 49, 39, 56, 34, 53,
       46, 42, 50, 36, 29, 32]

# Function to perform left shift according to LSH table
def left_shift(bits, n):
    return bits[n:] + bits[:n]

# Convert binary string to list of bits
def binary_to_bits(binary_string):
    return list(map(int, binary_string))

# Apply permutation according to PC1 table
def permute_key(key):
    return [key[i-1] for i in PC1]

# Apply permutation according to PC2 table
def permute_key_round(key):
    return [key[i-1] for i in PC2]

# Convert list of bits to binary string
def bits_to_binary(bits):
    return ''.join(map(str, bits))

# Generate round keys
def generate_round_keys(K0_binary):
    # Perform PC1 permutation to get K0
    K0_permuted = permute_key(K0_binary)
    C0 = K0_permuted[:28]  # Left half
    D0 = K0_permuted[28:]  # Right half

    round_keys = []
    C, D = C0, D0
    for i in range(16):
        # Perform left shift according to LSH table
        C = left_shift(C, LSH[i])
        D = left_shift(D, LSH[i])

        # Combine C and D
        CD = C + D

        # Perform PC2 permutation to get round key
        round_key = permute_key_round(CD)

        # Append round key to list
        round_keys.append(bits_to_binary(round_key))

        # Print C, D, and round key
        print(f"Iteration {i+1}:")
        print("C{}: {}".format(i+1, bits_to_binary(C)))
        print("D{}: {}".format(i+1, bits_to_binary(D)))
        print("K{}: {}".format(i+1, bits_to_binary(round_key)))
        print()

    return round_keys

def main():
    # Original 64-bit key
    K0_binary = '0100110001001111010101100100010101000011010100110100111001000100'

    # Generate round keys
    round_keys = generate_round_keys(binary_to_bits(K0_binary))

    # Print all round keys
    print("All Round Keys:")
    for i, key in enumerate(round_keys, 1):
        print("K{}: {}".format(i, key))

if __name__ == "__main__":
    main()
```

```
Iteration 1:
C1: 0000000111111110000000000100
D1: 1110110110011110100001101000
K1: 101000001001001011000010101100111110000011011110

Iteration 2:
C2: 0000001111111100000000001000
D2: 1101101100111101000011010001
K2: 101000000001001001010010110000110010111100011111

Iteration 3:
C3: 0000111111110000000000100000
D3: 0110110011110100001101000111
K3: 001001000101101001010000001111110011001110011000

Iteration 4:
C4: 0011111111000000000010000000
D4: 1011001111010000110100011101
K4: 000001100111000101010000011100010101000101100111

Iteration 5:
C5: 1111111100000000001000000000
D5: 1100111101000011010001110110
K5: 000011100100010101010001010001101010100010101110

Iteration 6:
C6: 1111110000000000100000000011
D6: 0011110100001101000111011011
K6: 010011110100000100001001111001000011110111011101

Iteration 7:
C7: 1111000000000010000000001111
D7: 1111010000110100011101101100
K7: 000010111000000110001001001010111001001011111011

Iteration 8:
C8: 1100000000001000000000111111
D8: 1101000011010001110110110011
K8: 000110010000100010001011010101111101110100100011

Iteration 9:
C9: 1000000000010000000001111111
D9: 1010000110100011101101100111
K9: 000110010000101010001000001111001101101100001110

Iteration 10:
C10: 0000000001000000000111111110
D10: 1000011010001110110110011110
K10: 000100000011100010001100110101000101010011110010

Iteration 11:
C11: 0000000100000000011111111000
D11: 0001101000111011011001111010
K11: 000100000010110001000100110011011010101001101001

Iteration 12:
C12: 0000010000000001111111100000
D12: 0110100011101101100111101000
```

```
K0_binary = '0100110001001111010101100100010101000011010100110100111001000100'
# round_keys = generate_round_keys(binary_to_bits(K0_binary))
# round_keys
# Define the E-bit selection table
E_BIT_SELECTION_TABLE = [32, 1, 2, 3, 4, 5,
                         4, 5, 6, 7, 8, 9,
                         8, 9, 10, 11, 12, 13,
                         12, 13, 14, 15, 16, 17,
                         16, 17, 18, 19, 20, 21,
                         20, 21, 22, 23, 24, 25,
                         24, 25, 26, 27, 28, 29,
                         28, 29, 30, 31, 32, 1]

# Define the S-boxes
S_BOXES = [
    # S1
    [[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
     [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
     [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
     [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]],
    # S2
    [[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],
     [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],
     [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],
     [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]],
    # S3
    [[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
     [13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
     [13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
     [1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]],
    # S4
    [[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],
     [13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
     [10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
     [3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]],
    # S5
    [[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
     [14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
     [4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
     [11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]],
    # S6
    [[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
     [10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
     [9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
     [4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]],
    # S7
    [[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],
     [13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
     [1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
     [6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]],
    # S8
    [[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
     [1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
     [7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
     [2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]]
]

# Define the P permutation table
P_PERMUTATION_TABLE = [16, 7, 20, 21, 29, 12, 28, 17,
                       1, 15, 23, 26, 5, 18, 31, 10,
                       2, 8, 24, 14, 32, 27, 3, 9,
                       19, 13, 30, 6, 22, 11, 4, 25]

# Define the initial permutation table (IP)
INITIAL_PERMUTATION_TABLE = [58, 50, 42, 34, 26, 18, 10, 2,
                             60, 52, 44, 36, 28, 20, 12, 4,
                             62, 54, 46, 38, 30, 22, 14, 6,
                             64, 56, 48, 40, 32, 24, 16, 8,
                             57, 49, 41, 33, 25, 17, 9, 1,
                             59, 51, 43, 35, 27, 19, 11, 3,
                             61, 53, 45, 37, 29, 21, 13, 5,
                             63, 55, 47, 39, 31, 23, 15, 7]

# Define the inverse initial permutation table (IP^-1)
INVERSE_INITIAL_PERMUTATION_TABLE = [40, 8, 48, 16, 56, 24, 64, 32,
                                     39, 7, 47, 15, 55, 23, 63, 31,
                                     38, 6, 46, 14, 54, 22, 62, 30,
                                     37, 5, 45, 13, 53, 21, 61, 29,
                                     36, 4, 44, 12, 52, 20, 60, 28,
                                     35, 3, 43, 11, 51, 19, 59, 27,
                                     34, 2, 42, 10, 50, 18, 58, 26,
                                     33, 1, 41, 9, 49, 17, 57, 25]


def f_function(R, K, index):
    # Apply E-bit selection table
    E_R = ''.join([R[i - 1] for i in E_BIT_SELECTION_TABLE])

    # XOR with the key
    B = bin(int(E_R, 2) ^ int(K, 2))[2:].zfill(48)

    # Split B into 8 parts of 6 bits each
    parts = [B[i:i + 6] for i in range(0, 48, 6)]

    # Apply S-boxes
    transformed bits = ''
```