Exercise on DES Encryption Techniques

February 6, 2018

Name: SONGA MUGABE Fabrice
Click HERE to go to the Link of the Code

1 DES Algorithm

```
# Permutation choice 1 made on the key O/P is 64 bits
CP_1 = [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]

# Matrix that determine the shift for each round of keys
SHIFT = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]

# Permutation choice 2 applied on shifted key to get Ki+1 (to produce 48 bits)
CP_2 = [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]

# Initial permutation matrix for the data IP Table O/P is 64 bits
PI = [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]

# E is TRUE if we are encrypting, else FALSE
# Expand matrix to get a 48bits matrix of data to apply the xor with Ki
E = [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]

# SBOX = 8 * [64 * [0]]
S_BOX = [

    [[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],
     ],

    [[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],
     ],

    [[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],
     ],

    [[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],
     ],

    [[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],
     ],

    [[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],
     ],

    [[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],
     ],

    [[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],
     ]
]

# Permutation made after each SBox substitution for each round,
# and the 32-bit half-block is expanded to 48 bits.
P = [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]

# Final permutation or Inverse permutation for initial permutation data after the 16 rounds
# The final permutation is the inverse of the initial permutation; the table is interpreted similarly
PI_1 = [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]

# Convert a string into a list of bits
def string_to_bit_array(plaintext):
    array = list()
    for char in plaintext:
        binval = binvalue(char, 8)         # Get the char value on one byte
        array.extend([int(x) for x in list(binval)])         # Add the bits to the final list
    return array

# Recreate the string from the bit array
def bit_array_to_string(array):
    res = ''.join([chr(int(y, 2))
                   for y in [''.join([str(x) for x in bytes]) for bytes in nsplit(array, 8)]])
    return res

# Return the binary value as a string of the given size
def binvalue(val, bitsize):
    bin_val = bin(val)[2:] if isinstance(val, int) else bin(ord(val))[2:]
    if len(bin_val) > bitsize:
        raise Exception("The binary value is larger than the expected size")
    while len(bin_val) < bitsize:
        bin_val = "0" + bin_val            # Add as many 0 as needed to get the wanted size
    return bin_val

# Split a list into n sizes sublists
def nsplit(s, n):
    return [s[k:k + n] for k in xrange(0, len(s), n)]

ENCRYPT = 1
DECRYPT = 0
```

```
class des():
    def __init__(self):
        self.password = None
        self.plaintext = None
        self.keys = list()

    def run(self, key, plaintext, action=ENCRYPT, padding=False):
        if len(key) < 8:
            raise Exception("The Key should be 8 bytes long")
        elif len(key) > 8:
            key = key[:8]              # If key size is above 8bytes, cut to be 8bytes long

        self.password = key
        self.plaintext = plaintext

        if padding and action == ENCRYPT:
            self.addPadding()
        elif len(self.plaintext) % 8 != 0:   # If not padding data size must be multiple of 8 bytes
            raise Exception("The Data size should be multiple of 8")

        self.generate_keys()                              # Generate all the keys
        plaintext_blocks = nsplit(self.plaintext, 8)  # Split the text in blocks of 8 bytes so 64 bits
        result = list()
        for block in plaintext_blocks:                      # Loop over all the blocks of data
            block = string_to_bit_array(block)            # Convert the block in bit array
            block = self.permute(block, PI)               # Apply the initial permutation
            g, d = nsplit(block, 32)                      # g(LEFT), d(RIGHT)
            tmp = None
            for i in range(16):                           # Do the 16 rounds
                d_e = self.permute(d, E)                  # Expand d to match Ki size (48bits)
                if action == ENCRYPT:
                    tmp = self.xor(self.keys[i], d_e)     # If encrypt use Ki
                else:
                    tmp = self.xor(self.keys[15 - i], d_e)   # If decrypt start by the last key
                tmp = self.substitute(tmp)                  # Method that will apply the SBOXes
                tmp = self.permute(tmp, P)
                tmp = self.xor(g, tmp)
                g = d
                d = tmp
            result += self.permute(d + g, PI_1)  # Do the last permutation and append the result to resu
        final_res = bit_array_to_string(result)

        if padding and action == DECRYPT:
            return self.removePadding(final_res)  # Remove the padding if decrypt and padding is true
        else:
            return final_res              # Return the final string of data ciphered/deciphered

    # Substitute bytes using SBOX
    def substitute(self, d_e):
        subblocks = nsplit(d_e, 6)        # Split bit array into sublist of 6 bits
        result = list()
        for i in range(len(subblocks)):          # For all the sublists
            block = subblocks[i]
            row = int(str(block[0]) + str(block[5]), 2) # Get the row with the first and last bit
```

```
            column = int(''.join([str(x) for x in block[1:][:-1]]), 2)  # Column is the 2,3,4,5th bits
            val = S_BOX[i][row][column]   # Take the value in the SBOX appropriated for the round (i)
            bin = binvalue(val, 4)           # Convert the value to binary
            result += [int(x) for x in bin]  # And append it to the resulting list
        return result

    # Permute the given block using the given table (so generic method)
    def permute(self, block, table):
        return [block[x - 1] for x in table]

    def xor(self, t1, t2):                       # Apply a xor and return the resulting list
        return [x ^ y for x, y in zip(t1, t2)]

    def generate_keys(self):                     # Algorithm that generates all the keys
        self.keys = []
        key = string_to_bit_array(self.password)
        key = self.permute(key, CP_1)            # Apply the initial permutation on the key
        g, d = nsplit(key, 28)                   # Split in the direction (g->LEFT),(d->RIGHT)
        for i in range(16):                      # Apply the 16 rounds
            g, d = self.shift_list(g, d, SHIFT[i]) # Apply the shift associated with the round (differen
            tmp = g + d                                 # Merging the two of them
            self.keys.append(self.permute(tmp, CP_2))   # Applying permutation to get the Ki

    def shift_list(self, g, d, n):                      # Shifting the list of the given value
        return g[n:] + g[:n], d[n:] + d[:n]

    def addPadding(self):                               # Add padding to the data using PKCS5 spec.
        pad_len = 8 - (len(self.plaintext) % 8)
        self.plaintext += pad_len * chr(pad_len)

    def removePadding(self, data):        # Removing the padding of the plaintext assumes there is padd
        pad_len = ord(data[-1])
        return data[:-pad_len]

    def encrypt(self, key, plaintext, padding=False):
        return self.run(key, plaintext, ENCRYPT, padding)


if __name__ == '__main__':
    key = [0x0F, 0x15, 0x71, 0xC9, 0x47, 0xD9, 0xE8, 0x59]
    print("The Key is : ", key)
    plaintext = [0x02, 0x46, 0x8A, 0xCE, 0xEC, 0xA8, 0x64, 0x20]
    print("The PlainText is : ", plaintext)
    d = des()
    r = d.encrypt(key, plaintext)
    print("The Cipher Text is : %r" %r)
```

2 The OUTPUT of the Program

```
/System/Library/Frameworks/Python.framework/Versions/2.7/
bin/python2.7 "/Users/admin/Dropbox/Practical/Data_sec/Lab 03/run.py"
('The Key is : ', [15, 21, 113, 201, 71, 217, 232, 89])
```

```
('The PlainText is : ', [2, 70, 138, 206, 236, 168, 100, 32])
The Cipher Text is : '\xda\x02\xce:\x89\xec\xac;'

Process finished with exit code 0
```