DES

Step 1. Define Lists, that Will Be Used in Calculations

The following lists should be hardcoded for further use in the DES encryption/decryption process:

- Permutation tables that are used to shift bits in particular order they work by shifting bit from their current position to the one that is the value of the index of permutation table (e.g. let's say that we have a bit with index 2, if the value in permutation table has value of 58, that means, that the bit value from original array is shifted from 2nd to 58th place and so on).

  Python implementation of permutation:

```
def __permutation(block, table):
    return [block[x - 1] for x in table]
```

  There are 3 permutation matrixes used in DES: Initial permutation (variable *PI*), Initial key permutation (variable *KP_1*) and secondary key permutation which is applied to shifted key (variable *KP_2*).

  Python implementation:

```
# Initial permutation matrix for the data blocks
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]

# Initial permutation made on the key
KP_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]

# Permutation applied on shifted key
KP_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]
```

- Expansion D-box, which is used to expand 32-bit matrix to get 48 bits in order to perform XOR operation with round key.

  Python implementation:

```
EXPANSION_DBOX = [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]
```

- S-boxes that are used in substitution operation in DES algorithm.
  Python implementation:

```
S_BOX = [
    # 1:
    [[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],
     ],
    # 2:
    [[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],
     ],
    # 3:
    [[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],
     ],
    # 4:
    [[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],
     ],
    # 5:
    [[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],
     ],
    # 6:
    [[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],
     ],
    # 7:
    [[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],
     ],
    # 8:
    [[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],
     ]
]
```

- Straight D-box, which is used to after each S-Box substitution for each round.
  Python implementation:

```
STRAIGHT_DBOX = [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 table, which is used after 16 DES rounds.
  Python implementation:
```
FINAL_PERMUTATION = [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]
```
- Shift array, which determines the bit shift for each round in key generation keys (1st, 2nd, 9th and 16th rounds are shifted by 1 bit, all others by 2 bits).
  Python implementation:
```
SHIFT = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]
```
  Python implementation of shift function:
```
def __shift(left, right, n):
    return left[n:] + left[:n], right[n:] + right[:n]
```

Step 2. Create Bit String

In this DES demonstration both key and message are required to be written in hex and later on they are converted to bit array which should be of length, that is multiple of 64 (if they are too short, 0 are added to the beginning). By default, it is believed that message and key will fit in 64 bits, but if they do not, the new closest length is calculated.

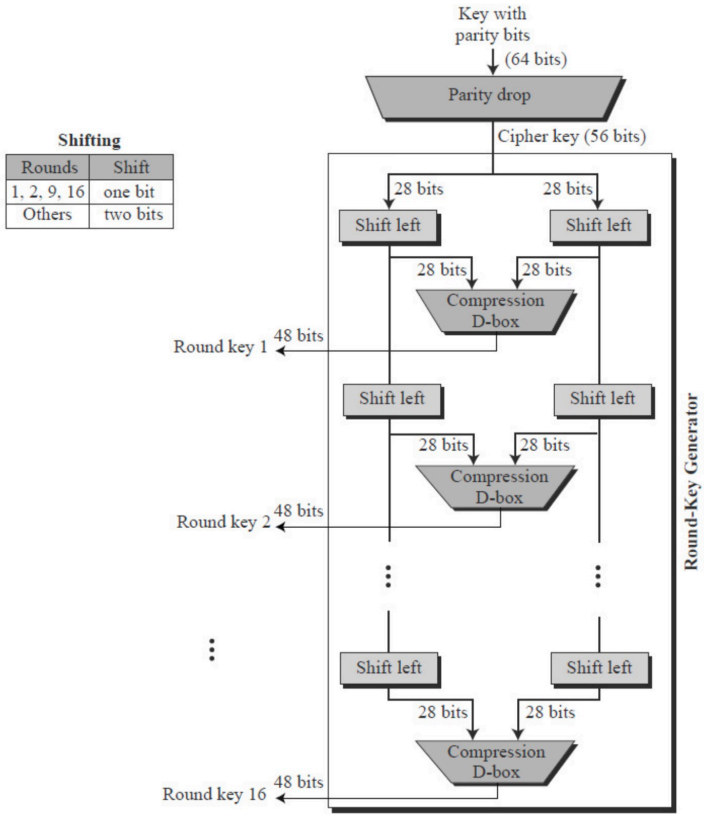
Python implementation:
```
@staticmethod
def __create_bit_array(string):
    # Convert array to 64 bit string
    bit_string = '{0:0{1}b}'.format(int(string, 16), 64)

    # Check if length of created string exceeds 64 bits:
    bit_string_len = len(bit_string)
    if bit_string_len > 64:
        # New length is calculated by multiplying 64 by sum of quotient of
current length divided by 64 and
        # integer value of boolean which checks if there is remainder
(which can be equal to 0 or 1)
        # This helps to find closest multiple of 64, for instance:
        # if the length is 128, it will remain the same (64 * (2 + 0)),
        # if the length is 129, it will become 192 (64 * (2 + 1))
        new_len = 64 * (int(bit_string_len / 64) + int(bit_string_len % 64
!= 0))

        # Define longer bit string:
        bit_string = '{0:0{1}b}'.format(int(string, 16), new_len)
    return [int(foo) for foo in bit_string]
```

Step 3. Generate Keys

Picture bellow demonstrates DES key generation steps. Initial value for round key generation is taken from main key after parity drop (application of Initial key permutation Key permutation table (variable *KP_1*)), then key is split in two equally sized blocks of 28 bits which get left shift according to round shift table (variable *SHIFT*), after that the shifted bits are put through Compression D-box (variable *KP_2*) and that's how the round key is extracted, the next round keys get initial value from the round key that was before.



Python implementation of key generation:

```
def __generate_keys(self):
    self.keys = []
    # Convert hex password to binary:
    key = self.__create_bit_array(self.password)
    # Perform initial permutation to a key (64bits -> 56 bits)
    key = self.__permutation(key, KP_1)
    # Split key in half:
    left = key[:28]
    right = key[28:]
    # Generate 16 keys
    for i in range(16):
        # Shift left and right sides by set steps in shift list
        left, right = self.__shift(left, right, SHIFT[i])
        temp = left + right  # merge two sides of a key
        # Perform final key permutation
        a = self.__permutation(temp, KP_2)
        self.keys.append(a)  # add to key list
```

Step 4. Define Substitution Function

Substitution in DES algorithm is quite simple: an array of 48 bits is passed and split into 8 sub arrays with 6 bits each, then every subarray is processed in such manner: the first and last bits are combined into an integer that represents a row number and the rest bits (2^{nd} to 4^{th}) represent the line number in particular S-box (variable *S_BOX*) according to the iteration number (going 1 through 8). The method then returns the array with respective values from the S-boxes that replace original values.

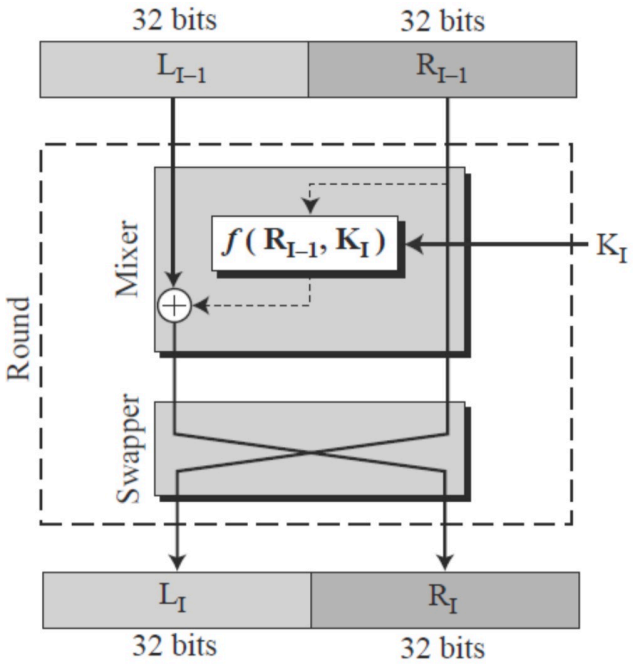
Python implementation:

```
def __substitute(self, right_expanded):
    # Split bit array into sublist of 6 bits
    subblocks = self.__split_array(right_expanded, 6)
    result = list()
    # For all the sub blocks:
    for i in range(len(subblocks)):
        block = subblocks[i]
        # Get decimal integer values for S-Box row and column:
        # Row is the first and last bits
        row = int(str(block[0]) + str(block[5]), 2)
        # Column consists of middle 4 bits (2nd to 5th)
        column = int(''.join([str(x) for x in block[1:][:-1]]), 2)
        # Take the value in the S-Box appropriated for the round (i)
        val = S_BOX[i][row][column]

        # Convert the value to binary and append it to the result
        result += [int(x) for x in str('{:04b}'.format(val))]
    return result
```

Step 5. Perform DES Rounds

Picture below illustrates DES round. First of all, message is split into blocks of 64 bits (if message is longer than that, every block is processed separately, and then final cipher is combined in the end).



DES round consists of following operations:

- Message (or 64-bit size message sized chunk if message itself is longer) undergoes initial permutation (Initial Permutation Table (variable *PI*) is applied). Then message blocks are split into equally sized (32 bit each) left and right blocks. These are initial values before performing 16 DES rounds for a message block.

  Python implementation:

  ```
  text_bits = self.__permutation(text_bits, PI)
  left = text_bits[:32]
  right = text_bits[32:]
  ```

- Message block is processed by 16 DES rounds with these procedures:
    - Right side is expanded to be 48-bit length by performing permutation with Expansion D-box (variable *EXPANSION_DBOX*).
      Python implementation:
      ```
      right_expanded = self.__permutation(right, EXPANSION_DBOX)
      ```
    - Action (encryption or decryption) is determined: if it's encryption, then keys are applied in normal order (from 1 to 16), in other case, keys are applied in inverse order (from 16 to 1). When action is determined, the XOR operation between correct key and expanded right (variable *right_expanded*) is performed. Then, the value (variable *temp*) is substituted with S-boxes and permutated with straight D-box. This is function *f*, which is illustrated above in the *Mixer* stage of DES round. The rest of Mixer stage is to apply XOR function with left side of message block and the result of previous operations (variable *temp*).
      Python implementation:
      ```
      right_expanded = self.__permutation(right, EXPANSION_DBOX)
      if action == 'encrypt':
          temp = self.__xor(self.keys[i], right_expanded)
      else:
          temp = self.__xor(self.keys[15 - i], right_expanded)
      temp = self.__substitute(temp)
      temp = self.__permutation(temp, STRAIGHT_DBOX)
      temp = self.__xor(left, temp)
      ```

      XOR implementation in Python:
      ```
      def __xor(t1, t2):
          return [int(x) ^ int(y) for x, y in zip(t1, t2)]
      ```
    - The swapper stage consists of switching left and right values.
      Python implementation:
      ```
      left = right
      right = temp
      ```
- After 16 rounds, final permutation is applied. Due to the fact that left and right sides are swapped in all rounds except for the last one, the function call for final permutation passes right first and then left, by doing this process is made faster, as there is no need to use if statement that checks round number during round application and makes calculations faster and more optimized.
  Python implementation:
  ```
  final_permutation = self.__permutation(right + left,
                      FINAL_PERMUTATION)
  ```
- Result of final permutation for a block is saved to a separate variable *result*, if it's the first block (value for the *result* variable is equal to *None*), the variable will get value of first block's final permutation result in case more blocks are present (message longer than 64 bits) so that they could be added to calculated result.
  Python implementation:
  ```
  result = final_permutation if result is None
           else result + final_permutation
  ```