Cryptography CSC 412/512
Programming Project

By: Reza Morovatdar

Student ID: 101071715

Email: reza.morovatdar@mines.sdsmt.edu

South Dakota Mines

Computer Science

December 2021

Contents

Introduction

This report is written to fulfill the programming project of the cryptography course CSC 412/512 taught by Dr. Christer Karlsson.

The first section covers the different classical cryptosystems, including Affine Cipher, Vigenere Cipher, etc. We provide codes for encoding, decoding, and attacks to these ciphers.

The second section is dedicated to basic number theory. Many useful functions for calculating great common divisor, extended Euclidian, the modular inverse of an integer, primitive root, the modular inverse of a matrix, primality check, random prime number, and factorization are represented.

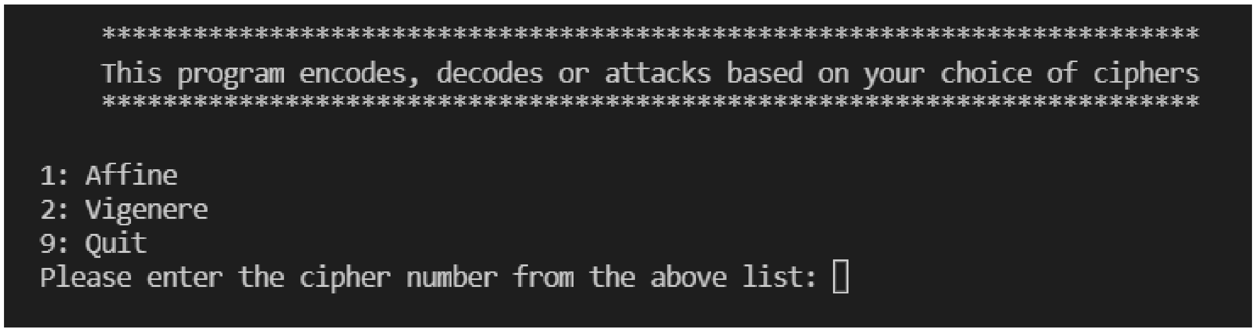
The third section covers Data Encryption Standard. With the implemented code, we can encrypt and decrypt with the simplified DES algorithm, do the three rounds and four rounds differential cryptanalysis, and encrypt and decrypt with the full 64-bit DES method.

The fourth part is RSA (Rivest–Shamir–Adleman) algorithm. As mentioned, we added primality check, random prime number, and factorization functions to our second section code. Using these tools and the extended Euclidian, we provided the implementation of RSA. We can encode and decode with the RSA algorithm.

Classical Cryptosystems

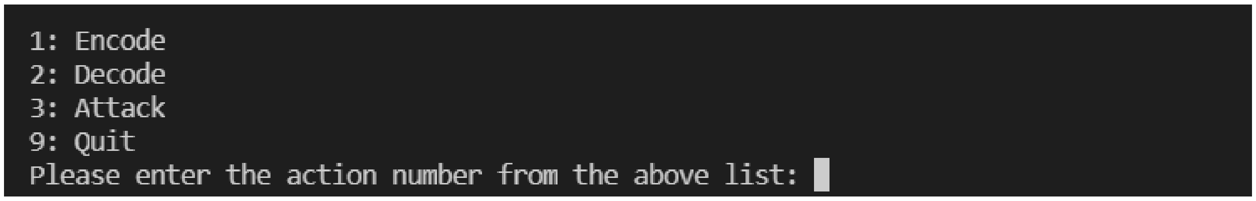
This part of the project is implemented in an interactive and user-friendly mode. The main module is *Classical.py*. It is connected to the cipher modules like *Affine.py*, *Vigenere.py,* etc.

By running *Classical.py,* the user has the option to choose the cipher methods as below:



```
    ************************************************************************
    This program encodes, decodes or attacks based on your choice of ciphers
    ************************************************************************


1: Affine
2: Vigenere
9: Quit
Please enter the cipher number from the above list: 
```

After choosing the cipher, the user can select the action they want to do with the selected cipher as below:



```
1: Encode
2: Decode
3: Attack
9: Quit
Please enter the action number from the above list: 
```

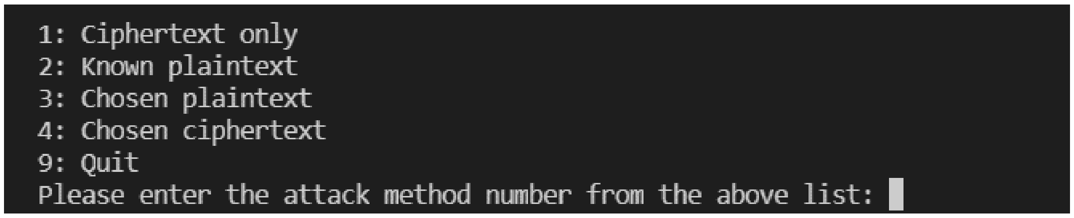
In the following subsections, we describe the theoretical background and implementation of each cipher:

Affine Ciphers

The affine cipher is a monoalphabetic substitution cipher, where each letter in an alphabet is mapped to its numeric equivalent, encrypted using a simple mathematical function, and converted back to a letter. The formula used means that each letter encrypts to one other letter and back again, meaning the cipher is essentially a standard substitution cipher with a rule governing which letter goes to which. As such, it has the weaknesses of all substitution ciphers. Each letter is enciphered with the function $\alpha x + \beta$ (mod 26), where β is the magnitude of the shift. And α should be chosen in a way that gcd(α,26) = 1, which means α cannot be 2, 13, or any multiple of those.

We implemented the encryption function in the *Affine.py* module by converting characters to numbers between 0 and 25 and calculating $\alpha x + \beta$ (mod 26). The decryption is also easy to perform using the *findModInverse* function, which will be described in the next section. Hence the decryption is done by calculating $\alpha^{-1}(x - \beta)$ (mod 26).

We have also covered all affine cipher attacks as shown below:



```
1: Ciphertext only
2: Known plaintext
3: Chosen plaintext
4: Chosen ciphertext
9: Quit
Please enter the attack method number from the above list: 
```

Chosen plaintext and chosen-ciphertext attacks are pretty straightforward and are implemented in the main menu of *Classical.py*. For the chosen plaintext attack, we feed the machine with the plaintext 'ab,' and for the chosen ciphertext, we ask the machine to decode ciphertext 'AB.' Then α and β can be easily calculated. On the other hand, known plaintext and Ciphertext only attacks can be tricky. We wrote independent functions for them in *Affine.py*.

The *KnownPlainAttack* function receives a list of plaintexts and their ciphertexts and finds the corresponding α and β keys by solving an equation in the *Solve* function.

CipherOnlyAttack function receives a long string of ciphertext. Using Frequency.py, we try to find the frequency of each letter and then estimate the high-frequency characters substitution to solve the corresponding equations (in the *Solve* function) and find α and β keys. We should notice that this function needs a very long string of ciphertext.

Vigenere Cipher

In the Vigenere cipher, each plaintext letter is shifted along different shift numbers in sequence. To encrypt, a keyword can be used. For example, suppose that the plaintext to be encrypted is "cryptography." The person sending the message chooses a keyword and repeats it until it matches the length of the plaintext, for example, the keyword "ENCODE": LEMONLEMONLE. Then the first letter of the plaintext, c, is paired with E, the first letter of the key so that c will be shifted four times (value of E). This method will continue for all characters of the plaintext, and finally, the ciphertext will be "GEADWSKECDKC."

The *Vignere.py* module includes three main functions to encode, decode, and attack. The *Encode* function can be easily done with the procedure mentioned above. The *Decode* function is also easy with the same procedure as described above but with the negative shifts along the keyword.

The attack is not that easy. The *Attack* function first computes (or it is better to say estimates) the length of the keyword and then uses the same approach as we used in the *CipherOnlyAttack* function of Affine cipher. It calculates the frequency of letters corresponding to each letter of the keyword and, based on the most frequent letters, tries to find that keyword's letter. Then it proceeds to the following letter to find the complete keyword.

Frequency Calculation

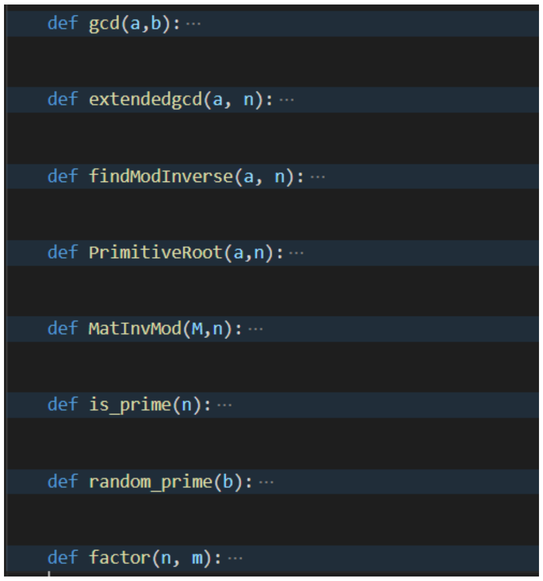
The *Frequency.py* module uses two inputs as follows:

1- *1frequency.txt*: The frequency table of single letters in English text.
2- *2frequency.txt*: The frequency table of letters pairs in English text.

By calculating the frequency of the letters in a provided text and comparing them with the frequency of letters (or pairs of letters) in the English language, we can better understand the ciphertext to try different kinds of frequency attacks.

Basic Number Theory

All the essential number theory tools needed for Cryptography are implemented in the *cryptomath.py* module. The list of the functions are as follows:

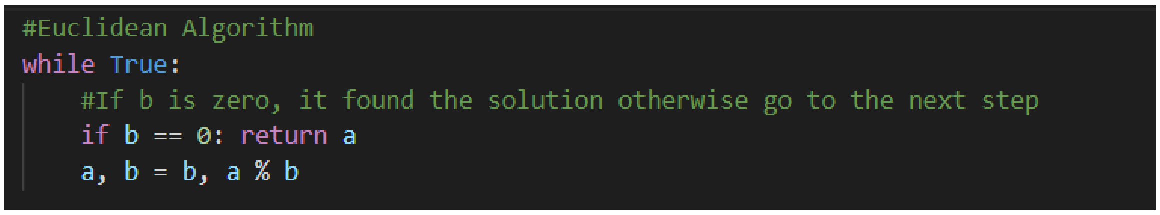


```
def gcd(a,b): ···

def extendedgcd(a, n): ···

def findModInverse(a, n): ···

def PrimitiveRoot(a,n): ···

def MatInvMod(M,n): ···

def is_prime(n): ···

def random_prime(b): ···

def factor(n, m): ···
```

We will discuss them one by one in the following subsections:

GCD

The great common divisor can be calculated easily by using the Euclidian algorithm. The euclidian algorithm is an efficient method for computing the greatest common divisor (GCD) of two integers (numbers), the largest number that divides them both without a remainder. The Euclidean algorithm is based on the principle that the greatest common divisor of two numbers does not change if the larger number is replaced by its difference with the smaller number. Since this replacement reduces the larger of the two numbers, repeating this process gives smaller pairs of numbers successively until the two numbers become equal. When that occurs, they are the GCD of the original two numbers. The respective iterative loop is as follows:



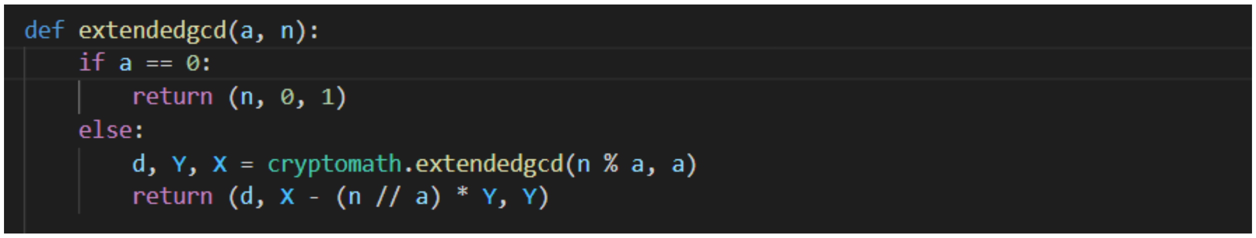
```
#Euclidean Algorithm
while True:
    #If b is zero, it found the solution otherwise go to the next step
    if b == 0: return a
    a, b = b, a % b
```

Extendedgcd

The extended Euclidean algorithm is the reverse steps of the Euclidean algorithm described above. By the extended Euclidean algorithm, the GCD can be expressed as a linear combination of the two original numbers, that is, the sum of the two numbers, each multiplied by an integer.

In other words, the extended Euclidean algorithm can find not only the GCD of numbers a and d but also integers X and Y in the equation $aX + bY = 1$.

The code presents two types of implementation of the extended Euclidean algorithm. The default code uses iterative function calls to calculate the GCD and return each step results in the reverse order to fulfill the extended part of the algorithm. The iterative function calls is as follows:



```python
def extendedgcd(a, n):
    if a == 0:
        return (n, 0, 1)
    else:
        d, Y, X = cryptomath.extendedgcd(n % a, a)
        return (d, X - (n // a) * Y, Y)
```

The second approach uses the standard forward and backward steps and extends the algorithm to the general case of solving $aX + bY = c$.

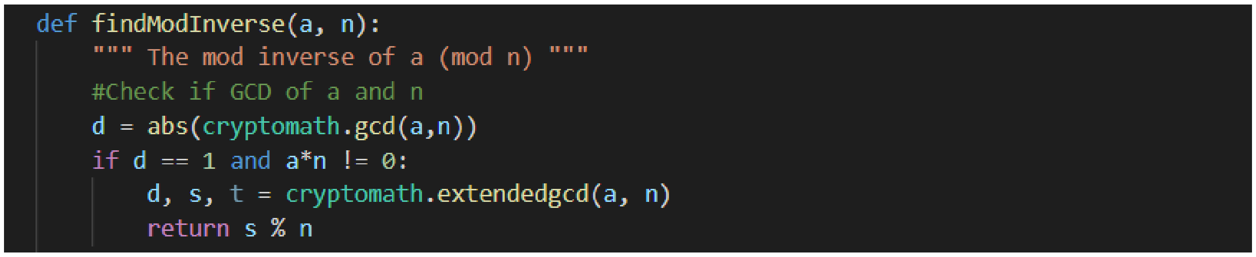
findModInverse

A modular multiplicative inverse of an integer a is an integer x such that the product ax is congruent to 1 (mod m). We can write the modular inverse as $ax \equiv 1\,(mod\,n)$ or $x \equiv a^{-1}\,(mod\,n)$.

For modular inverse to have a solution, the gcd of a and n should be 1. If a and n satisfies this condition, we can use the extended Euclidean algorithm to find the modular multiplicative inverse of a mod n. By the method described in the last section, we can deduce integers X and Y in a way that $ax + ny = 1$. It is equivalent to $ax - 1 = by$ or $ax \equiv 1\,(mod\,n)$.

The implemented function *findModInverse* calls the previous function *Extendedgcd* and return $x\,(mod\,n)$ as the modular multiplicative inverse of a.

The code is as follows:



```python
def findModInverse(a, n):
    """ The mod inverse of a (mod n) """
    #Check if GCD of a and n
    d = abs(cryptomath.gcd(a,n))
    if d == 1 and a*n != 0:
        d, s, t = cryptomath.extendedgcd(a, n)
        return s % n
```

PrimitiveRoot

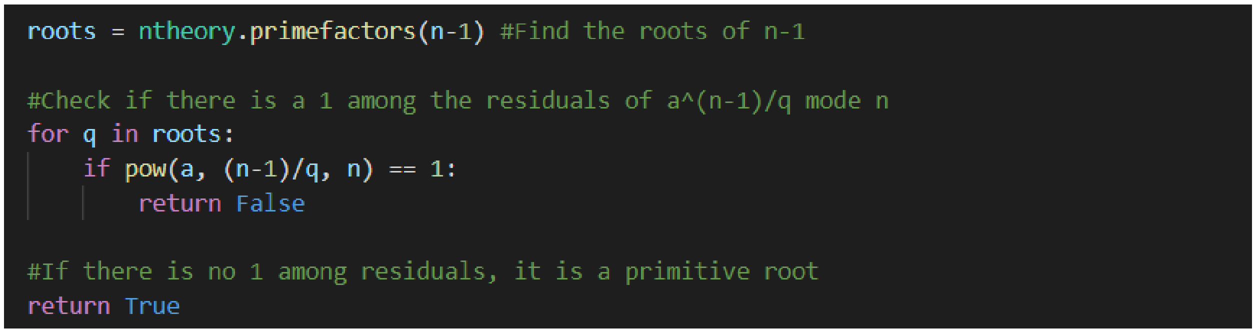
An integer a is a primitive root mod n if, for every integer c coprime to n, there is some integer k for which $a^k \equiv c\,(mod\,n)$.

To see if an integer n greater than 1 is a primitive root, first, compute $\varphi(n)$. Then determine the different prime factors of $\varphi(n)$, say $p_1, \ldots, p_k$. Finally, compute

$a^{\varphi(n)/p_i} \ (mod\ n)$ for $i = 1, \ldots, k$

The integer a for which these k results are all different from 1 is a primitive root.

The code is as follows:



```
roots = ntheory.primefactors(n-1) #Find the roots of n-1

#Check if there is a 1 among the residuals of a^(n-1)/q mode n
for q in roots:
    if pow(a, (n-1)/q, n) == 1:
        return False

#If there is no 1 among residuals, it is a primitive root
return True
```

MatInvMod

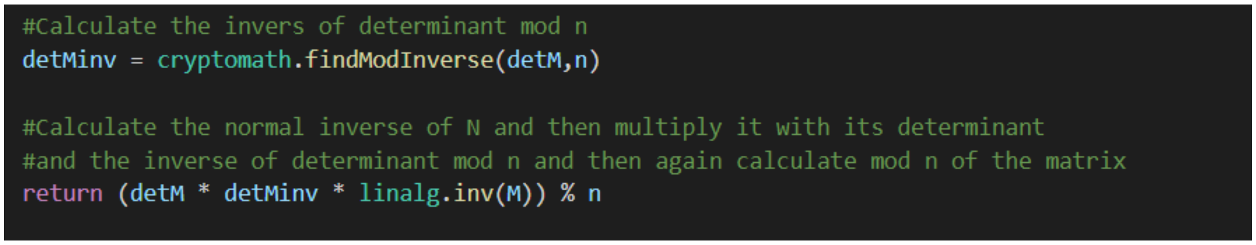
To calculate the inverse of matrix M mod n, first, we compute the determinant of the matrix, detM. If detM is coprime to n, then we can say M is invertible mod n. Then we can find the inverse of detA mod m with the algorithm described in the previous sections (*findModInverse* function). This we denote by detMinv and will be the unique integer between 0 and n which satisfies $(detM)\times(detM)^{-1} \equiv 1 \pmod{n}$. Next, we compute the matrix of cofactors of A, call this B. So, this is the matrix that would have been the usual inverse of M times the determinant. Or

$B = (detM)xM^{-1}$

The matrix $(detM)^{-1}\times B$ is the inverse to M (mod n) after calculating mod n of each matrix item. Hence,

$M^{-1} \pmod{n} \equiv (detM)^{-1}x(detM)xM^{-1} \pmod{n}$

The code was written as below:



```
#Calculate the invers of determinant mod n
detMinv = cryptomath.findModInverse(detM,n)

#Calculate the normal inverse of N and then multiply it with its determinant
#and the inverse of determinant mod n and then again calculate mod n of the matrix
return (detM * detMinv * linalg.inv(M)) % n
```

is_prime

There are different approaches for testing the primality of an integer n. The simplest method is Fermat Primality Test which chooses a random integer a between 1 and $n-1$ and calculates

$a^{n-1}\ (mod\ n)$. If it is not congruent to 1 (mod n), then n is composite, but if it is congruent to 1 (mod n), then n is probably prime. The Fermat Primality Test can result in wrong answers in rare cases, so we used a more strong approach for the primality test, Miller-Rabin Primality Test.

To test if n is prime with the Miller-Rabin Primality Test, we divide $n-1$ by two until an odd number m is reached. We say $n-1=2^k m$. We choose a random integer a between 1 and n– 1 and assign $b_0 \equiv a^m\ (mod\ n)$. If $b_0 \equiv \pm 1\ (mod\ n)$ then we say n is prime. Otherwise, we compute $b_i \equiv b_{i-1}^2\ (mod\ n)$ for i = 1, …, k-1. If in each iteration $b_i \equiv 1$, n is composite, if $b_i \equiv -1$, n is prime. Otherwise, continue the iteration. After the last iteration if $b_{k-1} \not\equiv -1$, then n is composite.

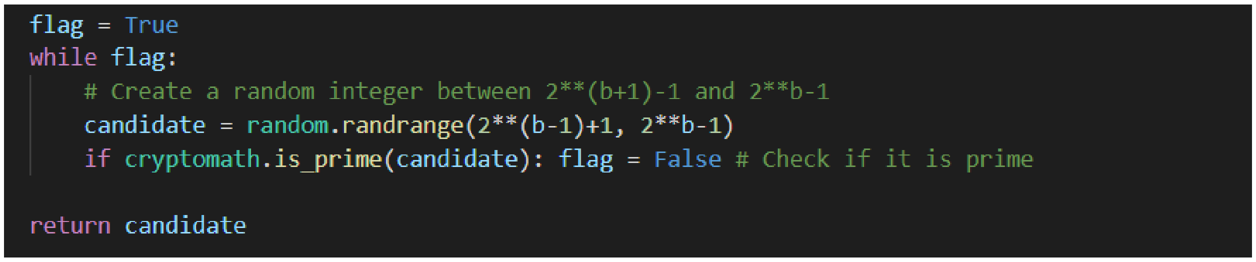
To be more confident about the result, we repeat the above algorithm 40 times with different random as. If in any iteration we find out n is composite then we stop the search and we conclude that n is composite, but if after complete calculation with 40 different random as, we can not show that n is composite then we conclude n is prime

random_prime

Generating a random prime between $2^{b+1}-1$ and 2^b-1 is an easy task. We use the *randrange* function from the *random* library of python to generate a random number in that range. Then we will check if the generated number is prime or not by using our *is_prime* function discussed in the previous section. If it is, we return the number. If not, we continue by generating a new random number in the mentioned range.

Since the density of primes around x is approximately $1/\ln x$, the probability of a random number between $2^{b+1}-1$ and 2^b-1 to be prime is between $1/(b+1)\ln 2$ and $1/(b)\ln 2$ or $\frac{1.44}{b+1}$ and $\frac{1.44}{b}$. Hence the prime number can be achieved in a short time.

The implemented code is as follows:



```
flag = True
while flag:
    # Create a random integer between 2**(b+1)-1 and 2**b-1
    candidate = random.randrange(2**(b-1)+1, 2**b-1)
    if cryptomath.is_prime(candidate): flag = False # Check if it is prime

return candidate
```

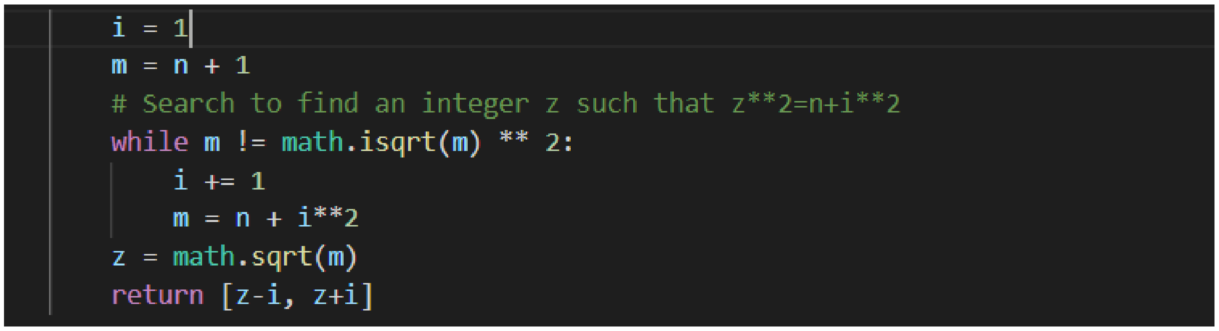
Factor

Three methods are used for factorizing an integer n. We will discuss each method here. Please note that we suppose n is the product of all methods of two prime numbers, p and q. Although if a factor of n is not prime, we can run the current factorization algorithm for that factor until all the prime factorization of n is reached.

Fermat's method

Fermat's factorization method is based on the representation of an odd integer as the difference of two squares, $n^2 = z^2 - i^2$ or $n^2 = (z-i)(z+i)$. To achieve this, we compute $n^2 + i^2$ for i from 1 to any number that makes $n^2 + i^2$ a complete square. Then z-i and z+i are two factors of n.

Fermat's method takes $|p-q|/2$ steps to complete, which means if the difference between p and q is large (or they are slightly different in length), finding the factors of n can be very expensive.



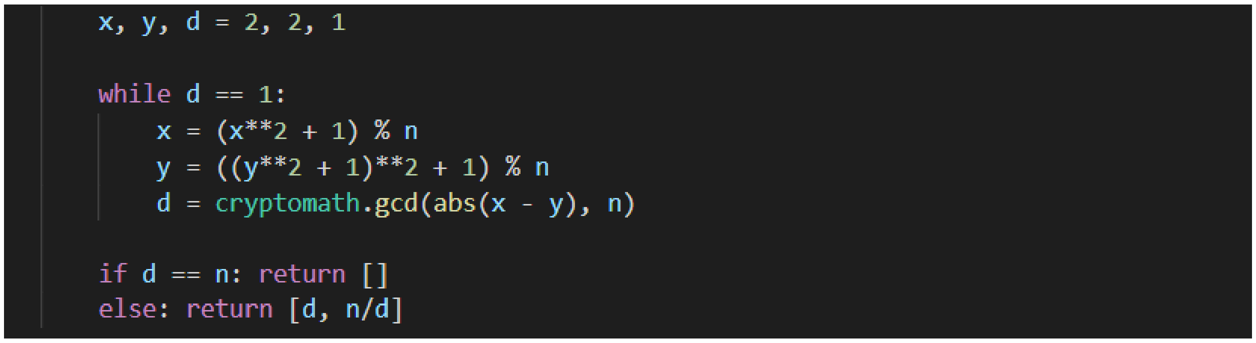
```
i = 1
m = n + 1
# Search to find an integer z such that z**2=n+i**2
while m != math.isqrt(m) ** 2:
    i += 1
    m = n + i**2
z = math.sqrt(m)
return [z-i, z+i]
```

Pollard rho

Pollard's rho algorithm uses $g(x)$ a polynomial mod n to generate a pseudorandom sequence. The starting value of x is selected randomly. For example, it can be $x_0 = 2$, then the sequence continues as $x_1 = g(2)$, $x_2 = g(g(2))$, $x_3 = g(g(g(2)))$ and so on. The sequence is related to another sequence $\{x_k \bmod p\}$. Since p is not known beforehand, this sequence cannot be explicitly computed in the algorithm. But, because the number of possible values for these sequences is finite, both the $\{x_k\}$ sequence, which is mod n, and $\{x_k \bmod p\}$ sequence will eventually repeat, even though these values are unknown. Once a sequence has a repeated value, the sequence will cycle because each value depends only on the one before it.

Two nodes x_i and x_j are kept. In each step, one moves to the next node in the sequence, and the other moves forward by two nodes. After that, it is checked whether gcd $(x_i - x_j, n) \neq 1$. If it is not 1, then this implies that there is a repetition in the $\{x_k \bmod p\}$ sequence, which means $x_i \pmod p \equiv x_j \pmod p$. The algorithm works because if the x_i congruent to x_j mod *p* then the difference between x_i and x_j is a multiple of *p*.

Pollard's rho algorithm sometimes fails to give the solution. This happens when $\gcd(x_i - x_j, n) = n$. In this case, we do the algorithm once again with a different initial number for x_0 or different $g(x)$ function.



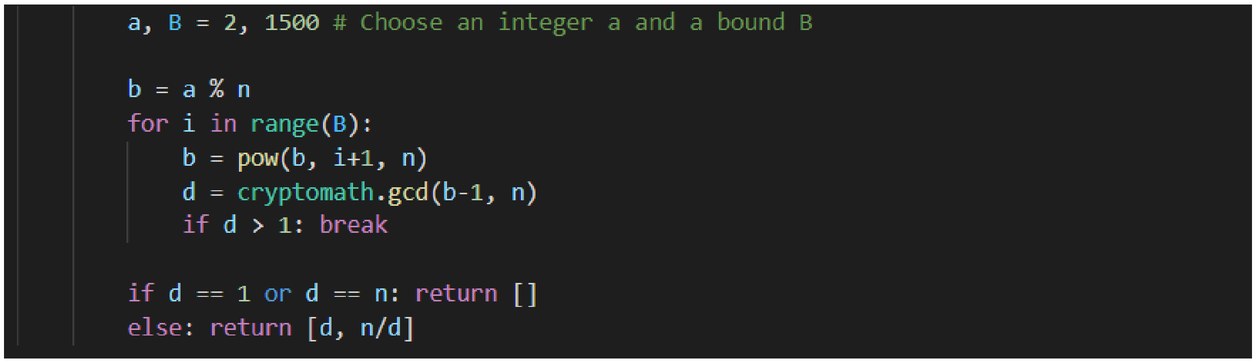
```
    x, y, d = 2, 2, 1

    while d == 1:
        x = (x**2 + 1) % n
        y = ((y**2 + 1)**2 + 1) % n
        d = cryptomath.gcd(abs(x - y), n)

    if d == n: return []
    else: return [d, n/d]
```

Pollard p-1

Pollard's p − 1 algorithm is designed for integers with specific types of factors. That is, if p is a factor of n, then p – 1 has only small prime factors. The existence of this algorithm leads to the concept of safe primes. It is necessary but insufficient that p − 1 has at least one large prime factor to be safe for cryptographic purposes.

To perform the algorithm, we choose an integer $b_1 = a > 1$ and a bound B. Then we compute $b_j \equiv b_{j-1}^{j} \pmod{n}$ and $d_j = \gcd(b_j - 1, n)$. If $1 < d_j < n$ a nontrivial factor of n is found. Otherwise, we will proceed to the next b_j and d_j. If *j* reaches the upper bound of B! and we do not find any factor for n, we will conclude that Pollard's p – 1 algorithm can not factor n or n does not have any factor that its p – 1 has only small prime factors.

The implementation of the algorithm is as follows:



```
    a, B = 2, 1500 # Choose an integer a and a bound B

    b = a % n
    for i in range(B):
        b = pow(b, i+1, n)
        d = cryptomath.gcd(b-1, n)
        if d > 1: break

    if d == 1 or d == n: return []
    else: return [d, n/d]
```

Data Encryption Standard

Data Encryption Standard (DES) is a symmetric key data encryption method adopted in 1977 for government agencies to protect sensitive data and was officially retired in 2005.

DES was developed at IBM and then submitted to the National Bureau of Standards (NBS) following the agency's invitation to propose a candidate to protect sensitive, unclassified electronic government data. In 1976, after consultation with the National Security Agency (NSA), the NBS selected a slightly modified version (strengthened against differential cryptanalysis, but weakened against brute-force attacks), which was published as an official Federal Information Processing Standard (FIPS) for the United States in 1977.

The effective DES key length of 56 bits would require a maximum of 256 or about 72 quadrillion attempts to find the correct key. This is not enough to protect data with DES against brute-force attempts with modern computers. DES remained a trusted and widely used encryption algorithm through the mid-1990s. However, in 1998, a computer built by the Electronic Frontier Foundation (EFF) decrypted a DES-encoded message in 56 hours. By harnessing the power of thousands of networked computers, EFF cut the decryption time to 22 hours the following year.

Some key features of DES are as follows:

64-bit block cipher: The Data Encryption Standard is a block cipher, meaning a cryptographic key and algorithm are applied to a 64-bit block of data simultaneously rather than one bit at a time.

16 rounds of encryption: The DES process involves encrypting via replacement and permutation 16 times.

64-bit key: DES uses a 64-bit key, but because eight of those bits are used for parity checks, the effective key length is only 56 bits.

DES was eventually replaced by the Advanced Encryption Standard (AES).

We implemented both the simplified DES and the full 64-bit DES as well as 3-rounds and 4-rounds differential cryptanalysis to attack the simplified DES.

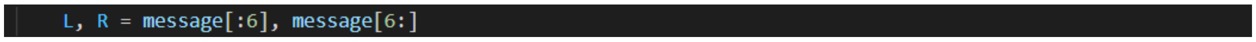
The modules are described in the following subsections.

Simplified DES

Like DES, the simplified DES algorithm is a block cipher. It works with 12-bit blocks of plaintexts and ciphertext, and the key size is 9-bit. There is no limitation on the number of rounds, but four rounds are typical to use.

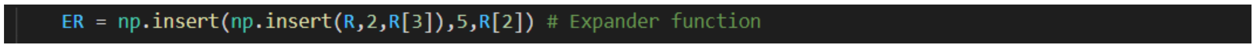
We implemented the above-mentioned simplified DES in *SimplifiedDES.py*. The algorithm consists of the following steps in each round:

1- Divide the block to 6-bit left and 6-bit right, L and R.



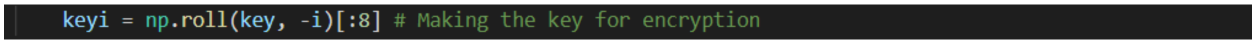
```
L, R = message[:6], message[6:]
```

2- Expand R from 6 bits to 8 bits.



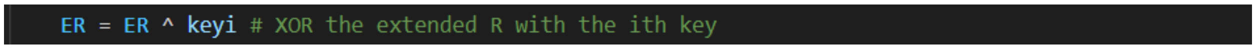
```
ER = np.insert(np.insert(R,2,R[3]),5,R[2]) # Expander function
```

3- Make the 8-bit ith round key by shifting the primary key i times and removing the last bit.



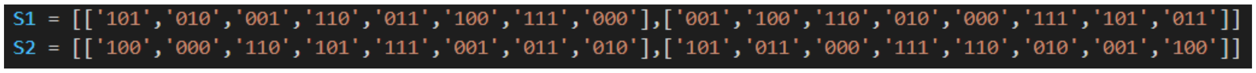
```
keyi = np.roll(key, -i)[:8] # Making the key for encryption
```

4- Calculate the XOR of the expanded R and the ith key.

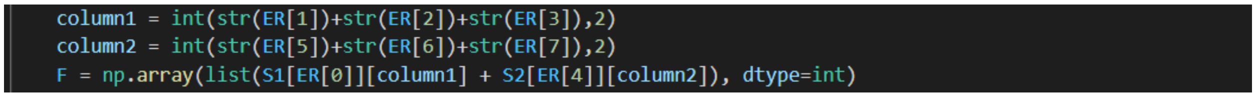


```
ER = ER ^ keyi # XOR the extended R with the ith key
```

5- Replace bits through two S-boxes and simultaneously convert the 8 bits of the previous step to 6 bits (the first S-box used for the first 4 bits and the second S-box for the last 4 bits).



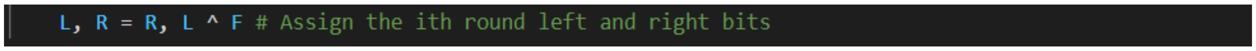
```
S1 = [['101','010','001','110','011','100','111','000'],['001','100','110','010','000','111','101','011']]
S2 = [['100','000','110','101','111','001','011','010'],['101','011','000','111','110','010','001','100']]
```



```
column1 = int(str(ER[1])+str(ER[2])+str(ER[3]),2)
column2 = int(str(ER[5])+str(ER[6])+str(ER[7]),2)
F = np.array(list(S1[ER[0]][column1] + S2[ER[4]][column2]), dtype=int)
```

6- Calculate the XOR of the previous step and L.

7- Assign the last step result to the new R and the last R to the new L.

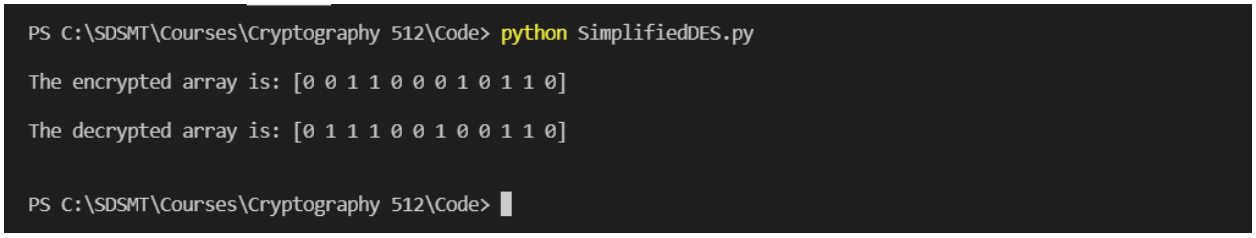


```
L, R = R, L ^ F # Assign the ith round left and right bits
```

8- Repeat the above steps for the intended number of rounds.

The decoding is also similar to the encoding with the same steps. We used the same function for both encoding and decoding, and the action will be given as an input argument to the function.

The key and the message block (as well as the cipher block) can be edited in the main part of the code. Running the encode and decode for key = '011001010' and message = '011100100110'



```
PS C:\SDSMT\Courses\Cryptography 512\Code> python SimplifiedDES.py

The encrypted array is: [0 0 1 1 0 0 0 1 0 1 1 0]

The decrypted array is: [0 1 1 1 0 0 1 0 0 1 1 0]


PS C:\SDSMT\Courses\Cryptography 512\Code> █
```

As shown, the SimplifiedDES can encrypt the message and decrypt it back to the original message.

Differential Cryptanalysis

Differential cryptanalysis is a general form of cryptanalysis applicable primarily to block ciphers. In the broadest sense, it is the study of how differences in information input can affect the resultant difference at the output. It refers to a set of techniques for tracing differences through the network of transformation, discovering where the cipher exhibits non-random behavior and exploiting such properties to recover the secret key (cryptography key). It should be noted that the first assumption of differential cryptanalysis is that we know all the inner workings of the algorithm except the key, and we want to determine the key.

3-round and 4-round differential cryptanalysis are simple forms of full cryptanalysis. 3-round cryptanalysis is a part of 4-round cryptanalysis, but it can also be implemented independently for a 3-round DES system. We implemented both methods in the *Cryptanalysis.py* module.

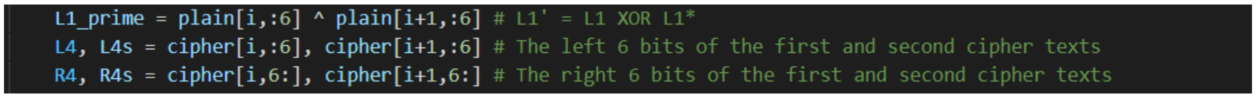
We describe each method in the following subsections.

Three-Round Differential Cryptanalysis

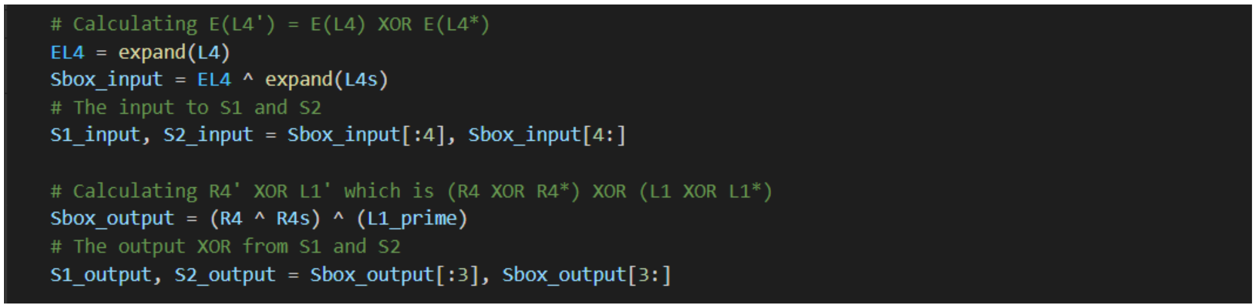
As mentioned, we eventually want to implement an attack to the Simplified DES when it uses four rounds, but we need to start by analyzing three rounds. Therefore, we use L_1R_1 instead of L_0R_0 to show the initial (plain) block. The algorithm has the following main steps:

0- Choose two plaintext blocks and their ciphertext blocks to perform differential cryptanalysis on them.

1- Calculate input XOR = $E(L_4')$ and output XOR = $R_4' \oplus L_1'$ for these two blocks

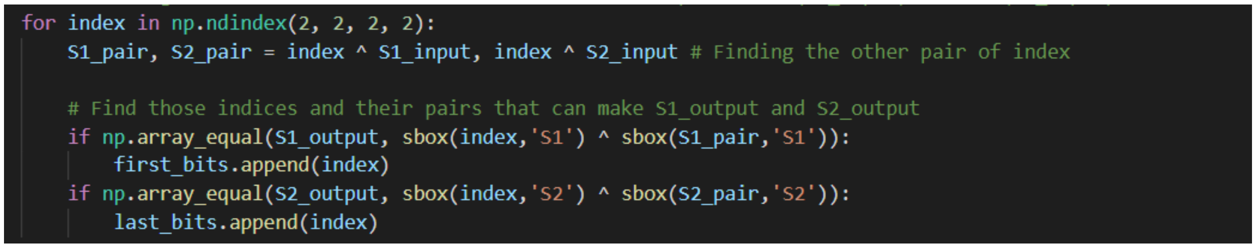


```
    L1_prime = plain[i,:6] ^ plain[i+1,:6] # L1' = L1 XOR L1*
    L4, L4s = cipher[i,:6], cipher[i+1,:6] # The left 6 bits of the first and second cipher texts
    R4, R4s = cipher[i,6:], cipher[i+1,6:] # The right 6 bits of the first and second cipher texts
```



```
    # Calculating E(L4') = E(L4) XOR E(L4*)
    EL4 = expand(L4)
    Sbox_input = EL4 ^ expand(L4s)
    # The input to S1 and S2
    S1_input, S2_input = Sbox_input[:4], Sbox_input[4:]

    # Calculating R4' XOR L1' which is (R4 XOR R4*) XOR (L1 XOR L1*)
    Sbox_output = (R4 ^ R4s) ^ (L1_prime)
    # The output XOR from S1 and S2
    S1_output, S2_output = Sbox_output[:3], Sbox_output[3:]
```

2. Look at the list of pairs with input XOR and output XOR and deduce the possibilities for K_4



```
for index in np.ndindex(2, 2, 2, 2):
    S1_pair, S2_pair = index ^ S1_input, index ^ S2_input # Finding the other pair of index

    # Find those indices and their pairs that can make S1_output and S2_output
    if np.array_equal(S1_output, sbox(index,'S1') ^ sbox(S1_pair,'S1')):
        first_bits.append(index)
    if np.array_equal(S2_output, sbox(index,'S2') ^ sbox(S2_pair,'S2')):
        last_bits.append(index)
```

3. The pair $(E(L_4) \oplus K_4,\ E(L_4^*) \oplus K_4)$ is on this list.

```
# XOR of E(L4) with the key (the 4th key)
if first_bits != []: first_bits = EL4[:4] ^ first_bits
if last_bits != []: last_bits = EL4[4:] ^ last_bits
```

4. Repeat until only one possibility for K_4 remains.

```
# Iterating between different pairs of plain texts and cipher texts
for i in range(len(plain)-1):
```

```
# Break the for loop, if only one candidate for the first 4 bits and one candidate for the last 4 bits remain
if left_candidates.ndim * right_candidates.ndim == 1: break
```

To test the module, we use the numbers in page 120 of the textbook as follows:

plaintext = ['000111011011','101110011011','010111011011']

ciphertext = ['000011100101','100100011000','001011001010']

We will have the same result that the textbook shows by running the code.

```
PS C:\SDSMT\Courses\Cryptography 512\Code> python Cryptanalysis.py

The key found from 3-round cryptanalysis is: [0 0 1 0 0 1 1 0 1]
```

Four-Round Differential Cryptanalysis

There is a weakness in S_1 the box. Looking at the 16 input pairs with XOR equal to 0011, we discover that 12 of them have output XOR equal to 011. There is a similar weakness in S_2, though not quite as extreme. Among the 16 input pairs with XOR equal to 1100, there are 8 with output XOR equal to 010. Assuming the outputs of the two S-boxes are independent, we see that the combined output XOR will be 011010 with probability (12/16)(8/16) = 3/8. When $L_1'R_1'$ = 001100000000, which should happen around 3/8 of the time, we try to use three-round differential cryptanalysis to find possible keys K_4. The list of many possible keys will contain K_4 and some other random keys. The remaining 5/8 of the time, the list should contain random keys. Since there seems to be no reason that any incorrect key should appear frequently, the correct key K_4 will probably appear in the lists of keys more often than the other keys.

The algorithm has the following steps:

1- Try several randomly chosen pairs of inputs with XOR equal to 011010001100 ($L_1'R_1'$). Look at the outputs L_4R_4 and $L_4^*R_4^*$.

```
# Random pair of inputs L0R0 and L0*R0* in a way that L0'R0'=011010001100
L0R0 = np.random.randint(2, size=12)
L0sR0s = L0R0 ^ [0,1,1,0,1,0,0,0,1,1,0,0]

L4R4, L4sR4s = encrypt(L0R0), encrypt(L0sR0s) # Calculate the cipher array L4R4 and L4*R4*
```

2- Use three-round differential cryptanalysis with L_1 = 001100 and the known outputs To deduce a set of possible keys K_4.

```
# Call the possible_keys function to find all possible first and last 4 bits of the 4th key
p_keys = possible_keys(L1_prime, L4R4[:6], L4R4[6:], L4sR4s[:6], L4sR4s[6:])
```

3- Select the more frequent key in the list of possible keys as the correct key K_4.

```
key = np.append(l_unq[np.argmax(l_cnt),:], r_unq[np.argmax(r_cnt),:]) # Find the most frequent key
```

To test the algorithm, we can denote a key to build the cipher machine. The key will not be forwarded to the *Cryptanalysis4* function, but the function calls the machine to look at the outputs of different inputs. We tried the example on page 123 of the textbook, and the algorithm correctly found the key.

```
The key found from 4-round cryptanalysis is: [1 0 1 1 1 0 0 0 0]


PS C:\SDSMT\Courses\Cryptography 512\Code> 
```

Full 64-DES

The full 64-bit DES is similar to the simplified DES in the algorithm's main steps.

We implemented the full 64-bit DES in *FullDES.py*. The algorithm consists of the following steps in each round:

1- Make an initial permutation

```
# Initial permutation of the message
message = permutate(message, IP)
```

1- Divide the block to 32-bit left and 32-bit right, L and R.

```
  L, R = message[:32], message[32:] # Dividing the message into two 32-bit left and right parts
```

2- Expand R from 6 bits to 8 bits.

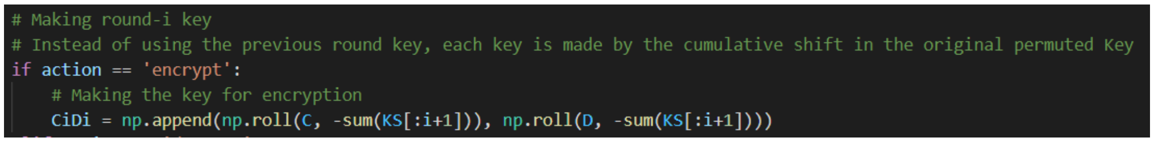
```
    ER = permutate(R, EP) # Expander function
```

3- Make the 56-bit ith round key by following steps:

3-1 First permutation of the key and divide the key block to 28-bit left and 28-bit right

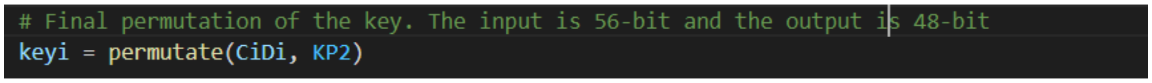
```
# Permutating the key which includes deleting the parity bits
# The input is 64-bit and the output is 56-bit
key = permutate(key, KP1)
C, D = key[:28], key[28:] # Dividing the key into two 28-bit left and right parts
```

3-2 shift the key with the cumulative amount of shifts up to round i.



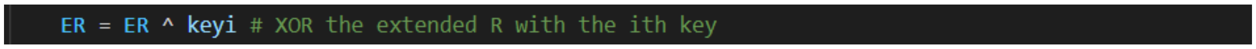
```
# Making round-i key
# Instead of using the previous round key, each key is made by the cumulative shift in the original permuted Key
if action == 'encrypt':
    # Making the key for encryption
    CiDi = np.append(np.roll(C, -sum(KS[:i+1])), np.roll(D, -sum(KS[:i+1])))
```

3-3 Second permutation of the key and covert the key to 48 bits.



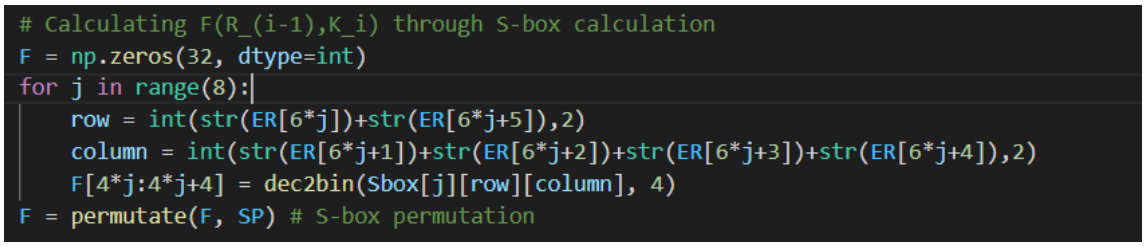
```
# Final permutation of the key. The input is 56-bit and the output is 48-bit
keyi = permutate(CiDi, KP2)
```

4- Calculate the XOR of the expanded R and the ith key.



```
    ER = ER ^ keyi # XOR the extended R with the ith key
```

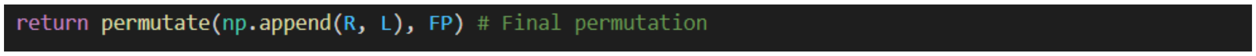
5- Run S-boxes.



```
# Calculating F(R_(i-1),K_i) through S-box calculation
F = np.zeros(32, dtype=int)
for j in range(8):
    row = int(str(ER[6*j])+str(ER[6*j+5]),2)
    column = int(str(ER[6*j+1])+str(ER[6*j+2])+str(ER[6*j+3])+str(ER[6*j+4]),2)
    F[4*j:4*j+4] = dec2bin(Sbox[j][row][column], 4)
F = permutate(F, SP) # S-box permutation
```

6- Calculate the XOR of the previous step and L.

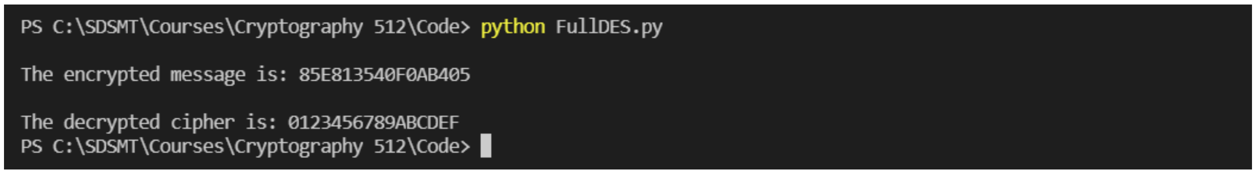
7- Assign the last step result to the new R and the last R to the new L.



```
return permutate(np.append(R, L), FP) # Final permutation
```

8- Repeat the above steps for the intended number of rounds.

Using the project description data, we ran the code, and the results show that the module works perfectly.



```
PS C:\SDSMT\Courses\Cryptography 512\Code> python FullDES.py

The encrypted message is: 85E813540F0AB405

The decrypted cipher is: 0123456789ABCDEF
PS C:\SDSMT\Courses\Cryptography 512\Code> 
```

RSA (Rivest–Shamir–Adleman)

RSA (Rivest–Shamir–Adleman) is an asymmetric and public key cryptographic algorithm. By asymmetric, we mean two different keys for encryption and decryption. RSA is called public-key because one of the keys can be given to anyone, and the other key must be kept private. The public key can be known to everyone and is used to encrypt messages. Messages encrypted using the public key can only be decrypted with the private key. The private key needs to be kept secret. Calculating the private key from the public key is very difficult. The algorithm is based on the fact that finding the factors of a large composite number is difficult. The algorithm can be summarized as follows:

1. Bob chooses secret primes p and q and computes n = pq.

2. Bob chooses e with gcd(e, (p - 1)(q - 1)) = 1.

3. Bob computes d with de ≡ 1 (mod (p - 1)(q - 1)).

4. Bob makes n and e public and keeps p, q, d secret.

5. Alice encrypts m as $c \equiv m^e$ (mod n) and sends c to Bob.

6. Bob decrypts by computing $m \equiv c^d$ (mod n).

As the generation of big primes is covered in the *cryptomath.py* module (*random_prime* function), we only cover steps 3 and 5 in the above summary to encrypt and decrypt a message by RSA given the chosen private keys.

To run the code, we used the data in the example of pages 165 and 166 of the textbook, as follows:

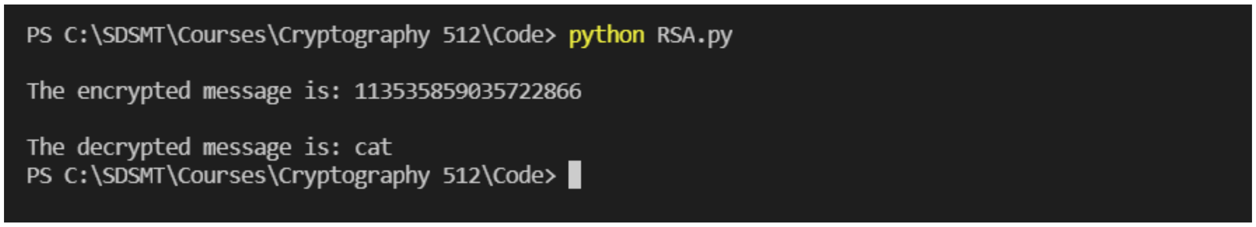
p = 885320963

q = 238855417

e = 9007

message = 'cat'

Running the code has the following results, which show that the algorithm is working correctly.



```
PS C:\SDSMT\Courses\Cryptography 512\Code> python RSA.py

The encrypted message is: 113535859035722866

The decrypted message is: cat
PS C:\SDSMT\Courses\Cryptography 512\Code> 
```