TABLE OF CONTENTS

S.NO	DATE	TITLE OF THE PROGRAM	MARKS	SIGN
1		Implementation of Caesar Cipher		
2		Basic Mono alphabetic Cipher		
3		Message Authentication Code		
4		Data Encryption Standard		
5		Advanced Encryption Standard		
6		Asymmetric Key Encryption		
7		Secure Key exchange		
8		Digital Signature Generation		
9		Implementation of Mobile Security		
10		Intrusion Detection/Prevention System with Snort		
11		Defeating Malware - Building Trojans		
12		Defeating Malware - Rootkit Hunter		
13		Implement Database Security		
14		Implement Encryption and Integrity Control-Database Security		

EX.NO:1	**Implementation of Caesar Cipher**

Aim:

To implement a Caesar cipher, a type of substitution cipher, that replaces each letter in a message with another letter based on a fixed shift value.

Algorithm:

1) Define Function:
   Define a function encrypt_text(plaintext, n) which takes plaintext (the text to be encrypted) and n (the shift pattern) as input.

2) Initialize Answer:
   Initialize an empty string ans to store the encrypted text.

3) Iterate Over Plaintext:
   Loop through each character ch in plaintext using its index i.

4) Check Character Type:
   If ch is a space, append a space to ans.
   Else if ch is an uppercase letter:
   Convert ch to its corresponding encrypted character using the formula:
   encrypted_char $==((\text{ord}(ch)+n-65)\%26)+65$
   Append the encrypted character to ans.
   Else if ch is a lowercase letter:
   Convert ch to its corresponding encrypted character using the formula:
   encrypted_char $=((\text{ord}(ch)+n-97)\%26)+97$
   Append the encrypted character to ans.

5) Return Encrypted Text:
   Return the encrypted text stored in ans.
6) Print Results:
   Define plaintext and n.
   Print the original plaintext and shift pattern.
   Call encrypt_text(plaintext, n) and print the resulting encrypted text.

Example

Given the plaintext = "HELLO EVERYONE" and n = 1:
Define plaintext and n.
Initialize ans as an empty string.
Loop through each character in "HELLO EVERYONE":
'H' is uppercase, so encrypt it to 'I'.
'E' is uppercase, so encrypt it to 'F'.
'L' is uppercase, so encrypt it to 'M'.
'L' is uppercase, so encrypt it to 'M'.

'O' is uppercase, so encrypt it to 'P'.
' ' is a space, so append ' '.
'E' is uppercase, so encrypt it to 'F'.
'V' is uppercase, so encrypt it to 'W'.
'E' is uppercase, so encrypt it to 'F'.
'R' is uppercase, so encrypt it to 'S'.
'Y' is uppercase, so encrypt it to 'Z'.
'O' is uppercase, so encrypt it to 'P'.
'N' is uppercase, so encrypt it to 'O'.
'E' is uppercase, so encrypt it to 'F'.
Return the encrypted text: "IFMMP FWFSZPOF".
Print the plaintext, shift pattern, and encrypted text.

Program:

Output:

Plain Text is: HELLO EVERYONE
Shift pattern is: 1
Cipher Text is: IFMMP FWFSZPOF

Result:

EX.NO:2	
	Basic Monoalphabetic Cipher

Aim:

To implements a basic monoalphabetic cipher, a type of substitution cipher, which replaces each letter in a message with another letter based on a fixed shift value.

Algorithm:

1) Define generate_cipher_key Function:
   - Input: shift (integer)
   - Initialize alphabet as a string containing 'abcdefghijklmnopqrstuvwxyz'.
   - Create shifted_alphabet by shifting alphabet by shift positions.
   - Create a dictionary key by mapping each character in alphabet to the
   - corresponding character in shifted_alphabet.
   - Return key.
2) Define encrypt Function:
   - Input: message (string), key (dictionary)
   - Initialize an empty string encrypted_message.
   - Loop through each character char in message:
   - If char is alphabetic:
   - If char is lowercase, append key[char] to encrypted_message.
   - If char is uppercase, append key[char.lower()].upper() to encrypted_message.
   - Else, append char to encrypted_message.
   - Return encrypted_message.
3) Define decrypt Function:
   - Input: ciphertext (string), key (dictionary)
   - Create reverse_key by reversing the key dictionary.
   - Initialize an empty string decrypted_message.
   - Loop through each character char in ciphertext:
   - If char is alphabetic:
   - If char is lowercase, append reverse_key[char] to decrypted_message.
   - If char is uppercase, append reverse_key[char.lower()].upper() to decrypted_message.
   - Else, append char to decrypted_message.
   - Return decrypted_message.
4) Define main Function:
   - Prompt user to input shift value.
   - Generate key using generate_cipher_key(shift).
   - Prompt user to choose between encryption and decryption (e or d).

- If the user chooses 'e':
- Prompt for the plaintext message.
- Encrypt the plaintext using encrypt(plaintext, key).
- Print the encrypted message.
- If the user chooses 'd':
- Prompt for the ciphertext message.
- Decrypt the ciphertext using decrypt(ciphertext, key).
- Print the decrypted message.
- If the user inputs an invalid choice, print an error message.

5) Execute main Function:

If this script is run as the main module, call the main function.

Program:

Output:

Enter the shift value for the cipher: 3
Encrypt or decrypt? (e/d): e
Enter the message to encrypt: hello world
Encrypted message: khoor zruog

Enter the shift value for the cipher: 3
Encrypt or decrypt? (d): d
Enter the message to decrypt: khoor zruog
Decrypted message: hello world

Result:

EX.NO:3	
	Message Authentication Code

Aim:

To calculate the messages digest of a text using the SHA-1, SHA-256, SHA-512 algorithm and thereby verifying data integrity.

Algorithm:

1) Import Hashlib Module:

   Import the hashlib module to use various SHA hash functions.

2) Compute SHA256 Hash:
   - Initialize a string str with the value "GeeksforGeeks".
   - Encode the string using str.encode() to convert it to bytes.
   - Pass the encoded string to hashlib.sha256() to compute the SHA256 hash.
   - Get the hexadecimal representation of the hash using result.hexdigest().
   - Print the message "The hexadecimal equivalent of SHA256 is :" followed by the hexadecimal value of the SHA256 hash.
3) Encode and hash using SHA512:
   - Reinitialize the string: str = "GeeksforGeeks"
   - Encode the string: encoded_str = str.encode()
   - Hash the encoded string using SHA512: result = hashlib.sha512(encoded_str)
   - Print the hexadecimal equivalent
4) Encode and hash using SHA1:
   - Reinitialize the string: str = "GeeksforGeeks"
   - Encode the string: encoded_str = str.encode()
   - Hash the encoded string using SHA1: result = hashlib.sha1(encoded_str)
   - Print the hexadecimal equivalent.

Program:

Output:
Input: GeeksforGeeks
The hexadecimal equivalent of SHA256 is :
f6071725e7ddeb434fb6b32b8ec4a2b14dd7db0d785347b2fb48f9975126178f
The hexadecimal equivalent of SHA512 is :
0d8fb9370a5bf7b892be4865cdf8b658a82209624e33ed71cae353b0df254a75db63d1baa35ad99f2
6f1b399c31f3c666a7fc67ecef3bdcdb7d60e8ada90b722
The hexadecimal equivalent of SHA1 is :
4175a37afd561152fb60c305d4fa6026b7e79856

Result:

EX.NO:4	Data Encryption Standard

Aim:

To implement a symmetric-key block cipher algorithm known as Data Encryption Standard (DES).

Algorithm:

1) **Hexadecimal to Binary Conversion (`hex2bin`)**:

- Initialize a dictionary that maps each hexadecimal digit to its 4-bit binary equivalent.
- Initialize an empty string for the binary result.
- For each character in the input hexadecimal string:
  - Append the corresponding binary string from the dictionary to the result.
- Return the binary result.

2) **Binary to Hexadecimal Conversion (`bin2hex`)**:

- Initialize a dictionary that maps each 4-bit binary string to its hexadecimal equivalent.
- Initialize an empty string for the hexadecimal result.
- For each group of 4 bits in the input binary string:
  - Append the corresponding hexadecimal character from the dictionary to the result.
- Return the hexadecimal result.

3) **Binary to Decimal Conversion (`bin2dec`)**:

- Initialize the decimal result to 0.
- For each bit in the input binary string, from least significant to most significant:
  - Multiply the bit by 2i2^i2i (where iii is the bit's position) and add to the decimal result.
- Return the decimal result.

4) **Decimal to Binary Conversion (`dec2bin`)**:

- Convert the decimal number to its binary representation using Python's `bin` function and remove the "0b" prefix.

- If the length of the binary result is not a multiple of 4, pad with leading zeros to make it a multiple of 4.

- Return the padded binary result.

5) **Permute Function (`permute`)**:

- Initialize an empty string for the permutation result.
- For each position in the permutation array:
  - Append the bit from the input string at the given position to the result.
- Return the permutation result.

6) **Left Shift Function (`shift_left`)**:

- For the specified number of shifts:
  - Perform a left circular shift on the input string.
- Return the shifted string.

7) **XOR Function (`xor`)**:

- Initialize an empty string for the XOR result.
- For each bit in the input strings:
  - Append the result of the XOR operation on the corresponding bits to the result.
- Return the XOR result.

8) **Encryption Function (`encrypt`)**:

- Convert the plaintext from hexadecimal to binary using `hex2bin`.
- Perform an initial permutation using the `initial_perm` table.
- Split the permuted text into left and right halves.
- For each of the 16 rounds:
  - Expand the right half from 32 to 48 bits using the `exp_d` table.
  - XOR the expanded right half with the round key.
  - Substitute the result using the S-boxes.
  - Perform a permutation using the `per` table.
  - XOR the result with the left half.
  - Swap the left and right halves, except in the final round.
- Combine the final left and right halves.
- Perform a final permutation using the `final_perm` table.
- Return the result as binary.

9) **Key Generation**:

- Convert the key from hexadecimal to binary using `hex2bin`.
- Perform a parity bit drop using the `keyp` table to get a 56-bit key.
- Split the key into left and right halves.
- For each of the 16 rounds:
  - Perform left shifts on both halves according to the `shift_table`.
  - Combine the left and right halves.
  - Compress the key from 56 to 48 bits using the `key_comp` table.

- Append the round key in both binary and hexadecimal form to the round key lists.

10) **Main Process**:

- Define the plaintext and key in hexadecimal format.
- Generate the round keys.
- Perform encryption using the generated round keys.
- Reverse the round keys for decryption.
- Perform decryption using the reversed round keys.

Program:

Output:

Plain Text: 123456ABCD132536
Key: AABB09182736CCDD
Encryption:
After initial permutation: 14A7D67818CA18AD
After splitting: L0=14A7D678 R0=18CA18AD
Round 1 18CA18AD 5A78E394 194CD072DE8C
Round 2 5A78E394 4A1210F6 4568581ABCCE
Round 3 4A1210F6 B8089591 06EDA4ACF5B5
Round 4 B8089591 236779C2 DA2D032B6EE3
Round 5 236779C2 A15A4B87 69A629FEC913
Round 6 A15A4B87 2E8F9C65 C1948E87475E
Round 7 2E8F9C65 A9FC20A3 708AD2DDB3C0
Round 8 A9FC20A3 308BEE97 34F822F0C66D
Round 9 308BEE97 10AF9D37 84BB4473DCCC
Round 10 10AF9D37 6CA6CB20 02765708B5BF
Round 11 6CA6CB20 FF3C485F 6D5560AF7CA5
Round 12 FF3C485F 22A5963B C2C1E96A4BF3
Round 13 22A5963B 387CCDAA 99C31397C91F
Round 14 387CCDAA BD2DD2AB 251B8BC717D0
Round 15 BD2DD2AB CF26B472 3330C5D9A36D
Round 16 19BA9212 CF26B472 181C5D75C66D
Cipher Text: C0B7A8D05F3A829C

Decryption
After initial permutation: 19BA9212CF26B472
After splitting: L0=19BA9212 R0=CF26B472
Round 1 CF26B472 BD2DD2AB 181C5D75C66D
Round 2 BD2DD2AB 387CCDAA 3330C5D9A36D
Round 3 387CCDAA 22A5963B 251B8BC717D0
Round 4 22A5963B FF3C485F 99C31397C91F

Round 5 FF3C485F 6CA6CB20 C2C1E96A4BF3
Round 6 6CA6CB20 10AF9D37 6D5560AF7CA5
Round 7 10AF9D37 308BEE97 02765708B5BF
Round 8 308BEE97 A9FC20A3 84BB4473DCCC
Round 9 A9FC20A3 2E8F9C65 34F822F0C66D
Round 10 2E8F9C65 A15A4B87 708AD2DDB3C0
Round 11 A15A4B87 236779C2 C1948E87475E
Round 12 236779C2 B8089591 69A629FEC913
Round 13 B8089591 4A1210F6 DA2D032B6EE3
Round 14 4A1210F6 5A78E394 06EDA4ACF5B5
Round 15 5A78E394 18CA18AD 4568581ABCCE
Round 16 14A7D678 18CA18AD 194CD072DE8C

Plain Text: 123456ABCD132536

Result:

EX.NO:5	Advanced Encryption Standard

Aim:

To understand the need of highly secured symmetric encryption algorithm known as Advanced Encryption Standard (AES)

Algorithm:

1. **Import Libraries:**
   - `AES` for AES encryption/decryption.
   - `get_random_bytes` to generate a random key.
   - `pad` and `unpad` to ensure data is of a valid block size.
2. **Encrypt Function:**
   - Creates a new AES cipher object in CBC mode.
   - Pads the data to be a multiple of the block size.
   - Encrypts the padded data.
   - Returns the initialization vector (IV) and the ciphertext.
3. **Decrypt Function:**
   - Creates a new AES cipher object with the same IV.
   - Decrypts the ciphertext.
   - Unpads and decodes the decrypted data.
4. **Example Usage:**
   - Generates a random 16-byte key.
   - Encrypts a sample message.
   - Prints the ciphertext in hexadecimal format.
   - Decrypts the ciphertext.
   - Prints the decrypted message.

1. **Encrypted Ciphertext:** The encrypted version of the plaintext message, displayed in hexadecimal format.
2. **Decrypted Data:** The original message after decrypting the ciphertext

Program:

The `pycryptodome` library in Python provides a simple way to implement AES.

pip install pycryptodome

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad
```

Sample Output:

Plain text: This is a secret message.
Key: b'?\xaa\xb2}m~\xa9\x08\xd4\x11\x10a\xec\xd0\xa5U'
Ciphertext: 0d82c44827bbe4bc68fe17df451f7a153c4f8bbb88769116cc7ff4582052921
Decrypted data: This is a secret message.

Result:

EX.NO:6	Asymmetric Key Encryption

Aim:

To implement the popular asymmetric key algorithm Rivest,Shamir ,Adleman (RSA)

Algorithm:

1) Input:
   - Two prime numbers p and q.
   - A plaintext message.
   - Calculate n:

     n = p * q
     For p=53 and q=59, n = 53 * 59 = 3127

2) Calculate the totient t:
   - t = (p - 1) * (q - 1)
   - For p=53 and q=59, t = (53 - 1) * (59 - 1) = 52 * 58 = 3016
   - Select the public key e:
   - Iterate from 2 to t to find the smallest integer e such that gcd(e, t) == 1.

3) Select the public key **e:**
   - Find the smallest integer e such that gcd(e, t) == 1
   - In this case, e = 3 (assuming the smallest integer that satisfies the condition)

4) Select the private key d:
   - Initialize j = 0.
   - Increment j in a loop until (j * e) % t == 1.
   - Set d = j.
   - Find d such that (d * e) % t == 1
   - Through iteration, if e = 3, then d = 2011 (assuming this is found through the while loop)

5) Encrypt the message:

   Calculate the ciphertext ct = (message ** e) % n.
   - ct = (message ** e) % n
   - For message=89, ct = (89 ** 3) % 3127 = 1394

6) Decrypt the message:
   - Calculate the decrypted message mes = (ct ** d) % n.
   - Print the encrypted message ct.

- Print the decrypted message mes
- mes = (ct ** d) % n
- For ct=1394, mes = (1394 ** 2011) % 3127 = 89

Program:

Output:

Output for Test Cases

Test Case 1

- Input: `p=53, q=59, message=89`
- Output:

  Encrypted message is 1394
  Decrypted message is 89

Test Case 2

- Input: `p=3, q=7, message=12`
- Output:

  Encrypted message is 3
  Decrypted message is 12

Result:

EX.NO:7	Secure Key exchange

Aim:

To securely exchange the crypto graphic keys over Internet to implement Diffie- Hellman key exchange mechanism

Algorithm:

1. **Input:**
    - p: A prime number.
    - g: A primitive root of p.
    - The user is prompted to enter a prime number p and a number g (which is a primitive root of p).
2. **Initialize Classes:**
    - **Class `A`:** Represents Alice and Bob.
        - `__init__`: Generate a random private number `n` for Alice/Bob.
        - `publish`: Calculate and return the public value `g^n % p`.
        - `compute_secret`: Compute the shared secret `(gb^n) % p` using another party's public value `gb`.
        - Represents Alice and Bob.
        - Generates a random private number n.
        - Computes and returns the public value using publish.
        - Computes the shared secret using compute_secret.
    - **Class `B`:** Represents Eve.
        - `__init__`: Generate two random private numbers `a` and `b` for Eve.
        - `publish`: Calculate and return the public value `g^arr[i] % p` for Eve's private numbers.
        - `compute_secret`: Compute the shared secret `(ga^arr[i]) % p` using another party's public value `ga`.
        - Represents Eve.
        - Generates two random private numbers a and b.
        - Computes and returns the public value using publish.
        - Computes the shared secret using compute_secret
3. **Create Instances:**
    - Create an instance of A for Alice.
    - Create an instance of A for Bob.
    - Create an instance of B for Eve.
    - Instances of A are created for Alice and Bob.

- An instance of B is created for Eve.
- Private numbers selected by Alice, Bob, and Eve are printed.
- Public values are generated and printed.
- Shared secrets are computed and printed.

4. **Print Private Numbers:**

   Print the private numbers selected by Alice, Bob, and Eve.

5. **Generate Public Values:**
   - Calculate Alice's public value ga = g^alice.n % p.
   - Calculate Bob's public value gb = g^bob.n % p.
   - Calculate Eve's public values gea = g^eve.a % p and geb = g^eve.b % p.
6. **Print Public Values:**

   Print the public values generated by Alice, Bob, and Eve.

7. **Compute Shared Secrets:**
   - Calculate Alice's shared secret with Eve sa = gea^alice.n % p.
   - Calculate Eve's shared secret with Alice sea = ga^eve.a % p.
   - Calculate Bob's shared secret with Eve sb = geb^bob.n % p.
   - Calculate Eve's shared secret with Bob seb = gb^eve.b % p.

8. **Print Shared Secrets:**

   Print the shared secrets computed by Alice, Bob, and Eve.

Program:

Output:
Enter a prime number (p) : 227
Enter a number (g) : 14
Alice selected (a) : 227
Bob selected (b) : 170

Eve selected private number for Alice (c) : 65
Eve selected private number for Bob (d) : 175
Alice published (ga): 14
Bob published (gb): 101

Eve published value for Alice (gc): 41
Eve published value for Bob (gd): 32
Alice computed (S1) : 41
Eve computed key for Alice (S1) : 41
Bob computed (S2) : 167
Eve computed key for Bob (S2) : 167

Result: