CRYPTOGRAPHY AND NETWORK SECURITY LAB MANUAL

List of Programs:

1. Write a C program that contains a string (char pointer) with a value ‘Hello world’. The program should XOR each character in this string with 0 and displays the result.
2. Write a C program that contains a string (char pointer) with a value ‘Hello world’. The program should AND or and XOR each character in this string with 127 and display the result.
3. Write a Java program to perform encryption and decryption using the following algorithms
    a. Ceaser cipher b. Substitution cipher c. Hill Cipher
4. Write a C/JAVA program to implement the DES algorithm logic.
5. Write a C/JAVA program to implement the Blowfish algorithm logic.
6. Write a C/JAVA program to implement the Rijndael algorithm logic.
7. Write the RC4 logic in Java Using Java cryptography; encrypt the text “Hello world” using Blowfish. Create your own key using Java key tool.
8. Write a Java program to implement RSA algorithm.
9. Implement the Diffie-Hellman Key Exchange mechanism using HTML and JavaScript.
10. Calculate the message digest of a text using the SHA-1 algorithm in JAVA.
11. Calculate the message digest of a text using the MD5 algorithm in JAVA.

1. Write a C program that contains a string (char pointer) with a value 'Hello world'. The program should XOR each character in this string with 0 and displays the result.

Consider a string with value 'Hello world'. XOR each character in this string with 0.
The binary **XOR** (exclusive **OR**) operation has two inputs and one output. It is like the **ADD** operation which takes two arguments (two inputs) and produces one result (one output).

The inputs to a binary **XOR** operation can only be **0** or **1** and the result can only be **0** or **1**. The binary **XOR** operation (also known as the binary **XOR** function) will always produce a **1** output if either of its inputs is **1** and will produce a **0** output if both of its inputs are **0** or **1**.
If we call the inputs **A** and **B** and the output **C** we can show the **XOR** function as:

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

XOR will be performed on Hello world and 0. The result obtained would be Hello world.

Expected Output:
string is
Hello world
after XOR
Hello world

2. Write a C program that contains a string (char pointer) with a value 'Hello world'. The program should AND or and XOR each character in this string with 127 and display the result.

Consider a string with value 'Hello world'. XOR each character in this string with 127.

The binary **XOR** (exclusive **OR**) operation has two inputs and one output. It is like the **ADD** operation which takes two arguments (two inputs) and produces one result (one output).

The binary **XOR** operation (also known as the binary **XOR** function) will always produce a **1** output if either of its inputs is **1** and will produce a **0** output if both of its inputs are **0** or **1**.
If we call the inputs **A** and **B** and the output **C** we can show the **XOR** function as:

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

XOR will be performed on Hello world and 127. The result obtained would be Hello world.

The binary **AND** operation will always produce a **1** output if both of its inputs are **1** and will produce a **0** output if either of its inputs are **0** .
If we call the inputs **A** and **B** and the output **C** we can show the **AND** function as:

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

Expected output:

modified string after AND operation on string and 127 is Hello World
modified string after performing XOR on string and 127 is 7-(

3. Write a Java program to perform encryption and decryption using the following algorithms

a. Ceaser cipher b. Substitution cipher c. Hill Cipher

a. Ceaser cipher

The Caesar cipher is one of the earliest known and simplest ciphers. It is a type of substitution cipher in which each letter in the plaintext is 'shifted' a certain number of places down the alphabet. For example, with a shift of 1, A would be replaced by B, B would become C, and so on. The method is named after Julius Caesar.

If the numbers 0 to 25 are assigned to alphabets then c = E(3, p) = (p+3) mod 26.

A shift of *k* characters is the general Caesar algorithm. Encryption and decryption formulas are given as:

c = E (*k*, p) = (p + *k*) mod 26, where *k*= 1 to 25

p = D (*k*, c) = (c-*k*) mod 26.

ALGORITHM:

Read the plain text from the user. Read the key value from the user.
If the key is positive then encrypt the text by adding the key with each character in the plain text.
Else subtract the key from the plain text.
Display the cipher text obtained above.

Expected Output:

Enter the Key: 2
Encrypted String is: Jgnnq
Decrypted String is: Hello

b. Substitution cipher

Substitution cipher is a method of encrypting by which units of plaintext are replaced with cipher text, according to a fixed system; the "units" may be single letters (the most common), pairs of letters, triplets of letters, mixtures of the above, and so forth. The receiver deciphers the text by performing the inverse substitution.

ALGORITHM:

Read the plain text from the user. Get substitution values from the user. Substitute the values.
Display the cipher text obtained above.

Expected Output:

Enter any string:
hello
string is:hello
The encrypted data is: svool
The decrypted data is: hello

c. Hill Cipher

The Hill cipher is a substitution cipher invented by Lester S. Hill in 1929. Each letter is represented by a number modulo 26. To encrypt a message, each block of n letters is multiplied by an invertible n × n matrix, again modulus 26.

To decrypt the message, each block is multiplied by the inverse of the matrix used for encryption. The matrix used for encryption is the cipher key, and it should be chosen randomly from the set of invertible n × n matrices (modulo 26).

The formula is c = k * p (mod 26), where k is m × m matrix (entries are mod26). p, c are column vectors of size m. To get plain text back we use the formula k^{-1}*c (mod 26) = p.

ALGORITHM:

Read the plain text and key from the user. Split the plain text into groups of length three. Arrange the keyword in a 3*3 matrix.
Multiply the two matrices (i.e plain text and key) and perform mod26 to obtain the cipher text of length three. Combine all these groups to get the complete cipher text.
In order to decrypt cipher text into plain text, find inverse of key matrix and multiply it with cipher text. Perform mod 26 on obtained matrix. Hence plain text is obtained.

Expected Output:

```
Enter 3x3 matrix for key (It should be inversible):
6   24 1
13 16 10
20 17 15
Enter a 3 letter string: cat
Encrypted string is : fin
Inverse Matrix is :
0.15873016     -0.7777778     0.50793654
0.011337869   0.15873016     -0.106575966
-0.2244898     0.85714287     -0.48979592
Decrypted string is : cat
```

4. Write a C/JAVA program to implement the DES algorithm logic.

DESCRIPTION:

The Data Encryption Standard (DES) is a symmetric-key block cipher. DES is an implementation of a Feistel Cipher. It uses 16 round Feistel structure. The block size is 64-bit. Though, key length is 64-bit, DES has an effective key length of 56 bits, since 8 of the 64 bits of the key are not used by the encryption algorithm (function as check bits only).

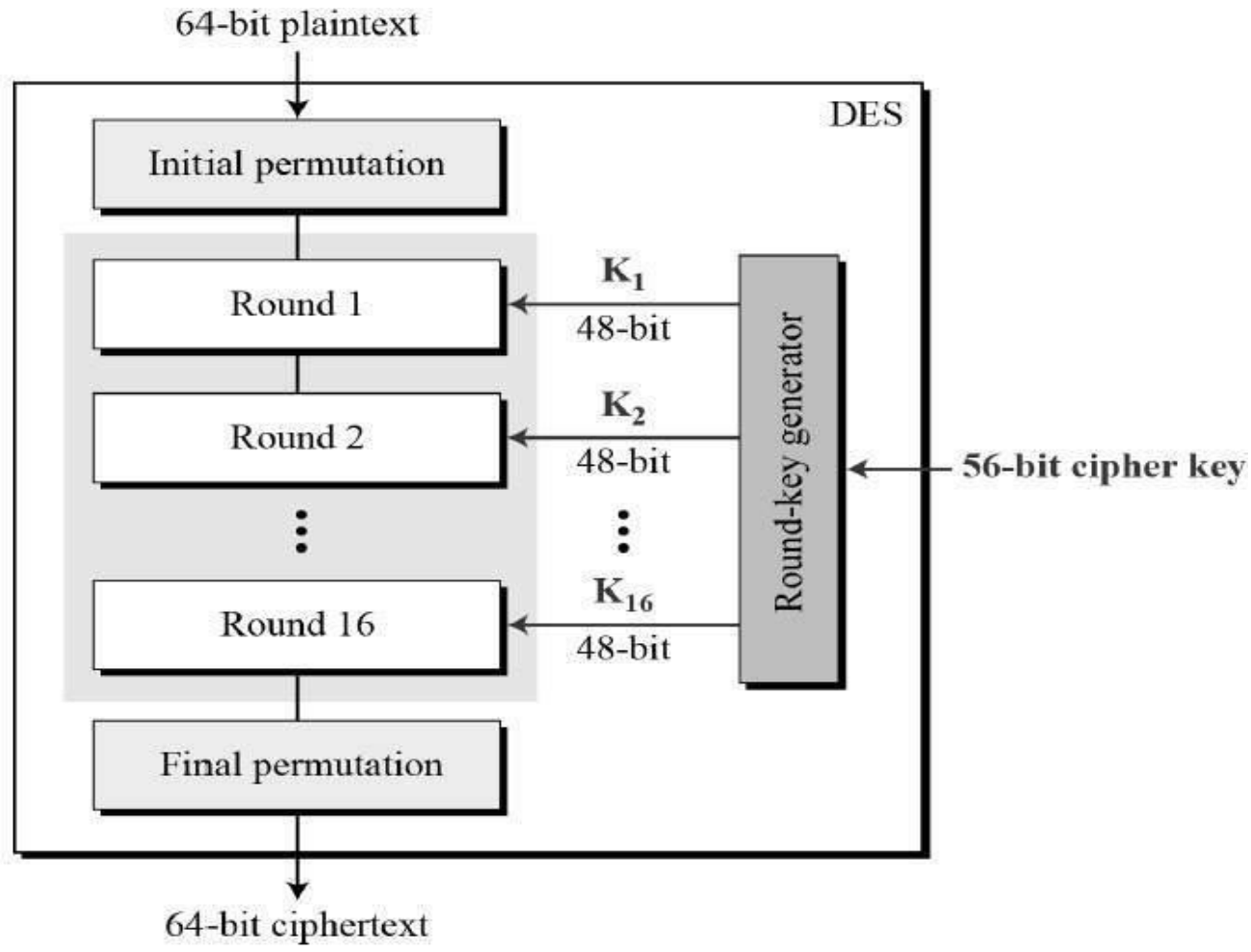
The algorithm involves carrying out combinations, substitutions and permutations between the text to be encrypted and the key, while making sure the operations can be performed in both directions.

ALGORITHM :

The main parts of the algorithm are as follows:

- Fractioning of the text into 64-bit blocks
- Initial permutation of blocks
- Breakdown of the blocks into two parts: left and right, named L and R
- Permutation and substitution steps repeated 16 times
- Re-joining of the left and right parts then inverse initial permutation

General Structure of DES is depicted in the following illustration



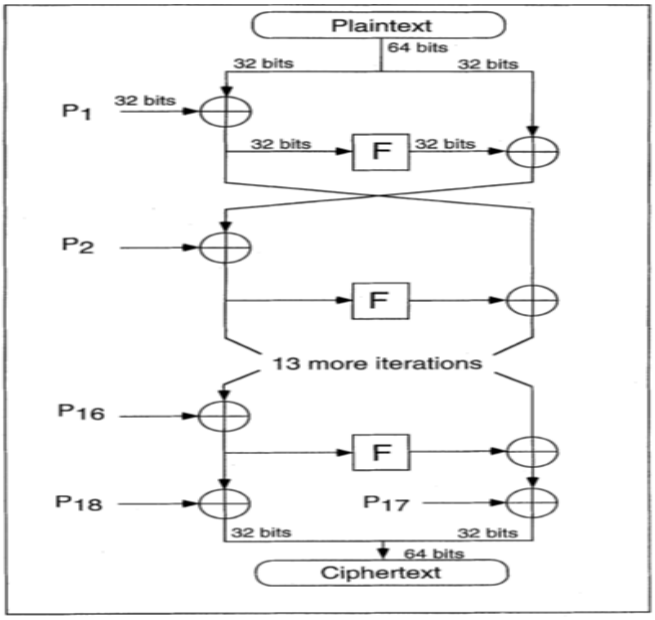
Expected Output:

Enter plain text: Madam
Key in hexadecimal used for encryption: 133457799BBCDFF1
Encrypted output: E5939798D7FB76E6
Decrypted output in hexadecimal: 6D6164616D202020
Decrypted output in plain text: madam

5. Write a C/JAVA program to implement the Blowfish algorithm logic.

DESCRIPTION:
Blowfish algorithm, it is a variable-length key block cipher. The block size is 64 bits, and the key can be any length up to 448 bits.
The algorithm consists of two parts a key-expansion part and a data-encryption part.
Key expansion converts a key of at most 448 bits into several subkey arrays totaling 4168 bytes.
Data encryption occurs via a 16-round Feistel network. Each round consists of a key dependent permutation, and a key- and data-dependent substitution. All operations are XORs and additions on 32-bit words. The only additional operations are four indexed array data lookups per round.

Algorithm: Blowfish Encryption
Divide x into two 32-bit halves: xL, xR
For i = 1to 16:
xL = XL XOR Pi
xR = F(XL) XOR xR
Swap XL and xR
Swap XL and xR (Undo the last swap.)
xR = xR XOR P17
xL = xL XOR P18
Recombine xL and xR



Expected Output:

Input your message: Hello World
Encrypted text: ?:RW--???S
Decrypted text: Hello World

6. Write a C/JAVA program to implement the Rijndael algorithm logic.

The algorithm Rijndael allows for a variety of block and key sizes and not just the 64 and 56 bits of DES' block and key size. The AES standard states that the algorithm can only accept a block size of 128 bits and a choice of three keys - 128, 192, 256 bits.

The algorithm begins with an **Add round key** stage followed by 9 rounds of four stages and a tenth round of three stages. This applies for both encryption and decryption with the exception that each stage of a round the decryption algorithm is the inverse of its counterpart in the encryption algorithm. The four stages are as follows:

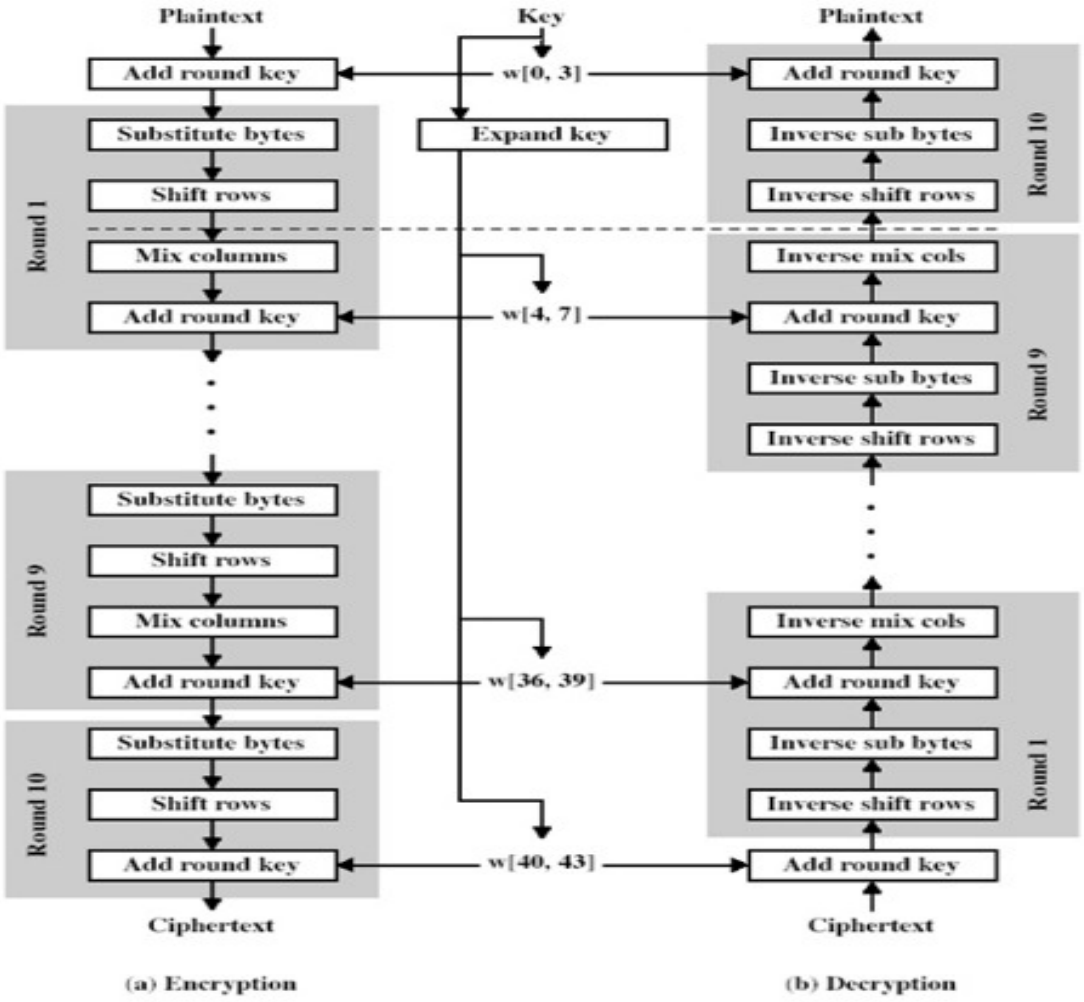
1. Substitute bytes
2. Shift rows
3. Mix Columns
4. Add Round Key

The tenth round simply leaves out the **Mix Columns** stage.

The first nine rounds of the decryption algorithm consist of the following:

1. Inverse Shift rows
2. Inverse Substitute bytes
3. Inverse Add Round Key
4. Inverse Mix Columns

Again, the tenth round simply leaves out the **Inverse Mix Columns** stage.



Expected Output:

Enter the length of Key(128, 192 or 256 only) 128
Enter the Key in hexadecimal
12 13 32 df 5d 4e c6 21 43 23 54 65 e3 d4 ed a2
Enter the CipherText in hexadecimal:
aa a3 c4 5d f2 12 97 67 ee cf a5 64 76 23 12 54
Text after decryption:
ec 23 80 c2 d4 01 a9 d4 03 3e 62 aa 3a 7d c6 57

7. Write the RC4 logic in Java Using Java cryptography; encrypt the text “Hello world” using Blowfish. Create your own key using Java key tool.

RC4 was designed in 1987 by Ron Rivest. RC4 is a stream cipher type. The RC4 algorithm has two phases, key generation, and encryption. Key generation is the first step and the most difficult in this algorithm. The encryption key is used to generate a variable encryption that uses two arrays, state and keys, and the results of merging operations. This merger operation consists of swapping, modulo, and other formulas. RC4 key generation is divided into several stages. Figure 2describes the stages of the RC4.S-Box initialization is to arrange the password occupied to be the byte array. Meanwhile, the permutation is to do the new byte array to as long as the plaintext available. The new key will be encrypted to the plaintext. It generates the cipher text.

RC4 generates a pseudo-random stream of bits (a key-stream). As with any stream cipher, these can be used for encryption by combining it with the plaintext using bit-wise exclusive-or. Decryption is performed the same way (since exclusive-or is a symmetric operation).

To generate the key stream, the cipher makes use of a secret internal state which consists of two parts:
1. A permutation of all 256 possible bytes (denoted "S" below).
2. Two 8-bit index-pointers (denoted "i" and "j").
The permutation is initialized with a variable length key, typically between 40 and 256 bits, using the key-scheduling algorithm (KSA). Then the stream of bits is generated by a pseudo-random generation algorithm.
Perform initialization of S
How it works sbox initialization, S [0], S [1], ..., S [255], with the numbers 0 to 255. First, the variable S will be filled with numbers from 0 to 255 in sequence S [0] = 0, S [1] = 1, ..., S[255] = 255. Then initialize another array, e.g., array K with a length of 256. The contents of the array K with a key that is repeated until the entire array K [0], K [1], ..., K [255] filled. S-Box initialization process is written as follows:

Keystream value search is done by exchanging again between elements S, but one value S stored in the K which is then used as a key stream. More details can be seen in the following pseudo code.

```
for i from 0 to 255
S[i] := i
endfor
j := 0
for i from 0 to 255
j := (j + S[i] + key[i
mod keylength]) mod 256
swap values of S[i] and S[j]
endfor
```

Expected Output:

RC4 Symmetric key = #□.,2B*
Encrypted message |&□pj□O□#□□
Decrypted message Hello World

8. Write a Java program to implement RSA algorithm.

The system was invented by three scholars **Ron Rivest, Adi Shamir,** and **Len Adleman** and hence, it is termed as RSA cryptosystem. Encryption and decryption are of the following form, for some plaintext block **M** and cipher text block **C**.

$$\mathbf{C}=M^e \bmod n$$
$$\mathbf{M}=C^d \bmod n$$

Both sender and receiver must know the value of n.
The sender knows the value of e, and only the receiver knows the value of d.
Thus, this is a public-key encryption algorithm with a public key PU = {e, n} and a private key PR = {d, n}. For this algorithm to be satisfactory for public-key encryption, the following requirements must be met.
1. It is possible to find values of e, d, n such that $M^{ed} \bmod n = M$ for all $M < n$.
2. It is relatively easy to calculate $M^e \bmod n$ and $C^d \bmod n$ for all values of $M < n$.
3. It is infeasible to determine d given e and n.

The preceding relationship holds if e and d are multiplicative inverses modulo φ(n), where φ(n) is the Euler totient function. It is shown that for p, q prime, φ (pq) = (p - 1)(q - 1). The relationship between e and d can be expressed as ed mod φ(n) = 1.
This is equivalent to saying
$ed \equiv 1 \bmod \varphi(n)$
$d \equiv e^{-1} \bmod \varphi(n)$

That is, e and d are multiplicative inverses mod φ (n). Note that, according to the rules of modular arithmetic, this is true only if d (and therefore e) is relatively prime to φ (n). Equivalently, gcd(φ (n), d) = 1.

Algorithm:

Read p, q that are prime numbers and p≠q.
Calculate n=p*q
Calculate ϕ(n) = (p-1)(q-1).
Select integer e that is prime to ϕ(n) and less than ϕ(n) i.e. select e, with gcd(φ (n),e)=1; 1 < e < φ (n)
Calculate $d=e^{-1} \bmod \varphi(n)$
Cipher text $C = M^e \bmod n$
Plain text $M = C^d \bmod n$
PU_A = {e, n}.
PR_A= {d, n}.

Expected Output:

Enter the number to be encrypted and decrypted: 4
Enter 1st prime number p 3
Enter 2nd prime number q 7
the value of z = 12
the value of e = 5
the value of d = 5
Encrypted message is : -
16.0
Decrypted message is : -
4.0

9. Implement the Diffie-Hellman Key Exchange mechanism using HTML and JavaScript.

Diffie–Hellman Key Exchange establishes a shared secret between two parties that can be used for secret communication for exchanging data over a public network. It is primarily used as a method of exchanging cryptography keys for use in symmetric encryption algorithms like AES. The algorithm in itself is very simple. The process begins by having the two parties, Alice and Bob. Let's assume that Alice wants to establish a shared secret with Bob.

ALGORITHM:

- Both Alice and Bob shares the same public keys g and p.
- Alice selects a random public key a.
- Alice computes his secret key A as ga mod p.
- Then Alice sends A to Bob.
- Similarly Bob also selects a public key b and computes his secret key as B and sends the same back to Alice.
- Now both of them compute their common secret key as the other one's secret key power of a mod p.

Global Public Elements:
Let q be a prime number and α where $\alpha < q$ and α is a primitive root of q.

1. User A Key Generation:
Select private X_A where $X_A < q$
Calculate public Y_A where $Y_A = \alpha^{XA} \bmod q$

2. User B Key Generation:
Select private X_B where $X_B < q$
Calculate public Y_B where $Y_B = \alpha^{XB} \bmod q$

3. Calculation of Secret Key by User A
$K = (Y_B)^{XA} \bmod q$

4. Calculation of Secret Key by User B:
$K = (Y_A)^{XB} \bmod q$

Expected Output:

simulation of Diffie-Hellman key exchange algorithm
aliceSends : 6.0
bobComputes : 9.0
bobSends : 5.0
aliceComputes : 9.0
sharedSecret : 9.0
success: shared secrets matches! 9.0

10. Calculate the message digest of a text using the SHA-1 algorithm in JAVA.

In cryptography, SHA-1 (Secure Hash Algorithm 1) is a cryptographic hash function. SHA-1 produces a 160-bit hash value known as a message digest. The way this algorithm works is that for a message of size < 264 bits it computes a 160-bit condensed output called a message digest. The SHA-1 algorithm is designed so that it is practically infeasible to find two input messages that hash to the same output message. A hash function such as SHA-1 is used to calculate an alphanumeric string that serves as the cryptographic representation of a file or a piece of data. This is called a digest and can serve as a digital signature. It is supposed to be unique and non-reversible.

ALGORITHM:

- Read the 256-bit key values.
- Divide into five equal-sized blocks named A, B, C, D and E.
- The blocks B, C and D are passed to the function F.
- The resultant value is permuted with block E.
- The block A is shifted right by 's' times and permuted with the result of step-4.
- Then it is permuted with a weight value and then with some other key pair and taken as the first block.
- Block A is taken as the second block and the block B is shifted by 's' times and taken as the third block.
- The blocks C and D are taken as the block D and E for the final output.

ALGORITHM DESCRIPTION:

Secured Hash Algorithm-1 (SHA-1):
Step 1: Append Padding Bits….
Message is "padded" with a 1 and as many 0's as necessary to bring the message length to 64 bits less than an even multiple of 512.
Step 2: Append Length....
64 bits are appended to the end of the padded message. These bits hold the binary format of 64 bits indicating the length of the original message.
Step 3: Prepare Processing Functions….
SHA1 requires 80 processing functions defined as:
f(t;B,C,D) = (B AND C) OR ((NOT B) AND D) (0 <= t <= 19)
f(t;B,C,D) = B XOR C XOR D (20 <= t <= 39)
f(t;B,C,D) = (B AND C) OR (B AND D) OR (C AND D) (40 <= t<=59)
f(t;B,C,D) = B XOR C XOR D (60 <= t <= 79)
Step 4: Prepare Processing Constants....
SHA1 requires 80 processing constant words defined as:
K(t) = 0x5A827999 (0 <= t <= 19)
K(t) = 0x6ED9EBA1 (20 <= t <= 39)
K(t) = 0x8F1BBCDC (40 <= t <= 59)
K(t) = 0xCA62C1D6 (60 <= t <= 79)
Step 5: Initialize Buffers….
SHA1 requires 160 bits or 5 buffers of words (32 bits):
H0 = 0x67452301
H1 = 0xEFCDAB89
H2 = 0x98BADCFE
H3 = 0x10325476
H4 = 0xC3D2E1F0

Step 6: Processing Message in 512-bit blocks (L blocks in total message)….
This is the main task of SHA1 algorithm which loops through the padded and appended message in 512-bit blocks.b
Input and predefined functions: M[1, 2, ..., L]: Blocks of the padded and appended message f(0;B,C,D), f(1,B,C,D), ..., f(79,B,C,D): 80 Processing Functions K(0), K(1), ..., K(79): 80 Processing Constant Words
H0, H1, H2, H3, H4, H5: 5 Word buffers with initial values
Step 7: Pseudo Code….

```
For loop on k = 1 to L
        (W(0),W(1),...,W(15)) = M[k] /* Divide M[k] into 16 words */
         For t = 16 to 79 do:
        W(t) = (W(t-3) XOR W(t-8) XOR W(t-14) XOR W(t-16)) <<< 1
        A = H0, B = H1, C = H2, D = H3, E = H4
        For t = 0 to 79 do:
        TEMP = A<<<5 + f(t;B,C,D) + E + W(t) + K(t) E = D, D = C,
        C = B<<<30, B = A, A = TEMP
        End of for loop
         H0 = H0 + A, H1 = H1 + B, H2 = H2 + C, H3 = H3 + D, H4 = H4 + E
 End of for loop
```

Expected Output:

Message digest object info:
Algorithm = SHA1
Provider = SUN version 1.8
ToString = SHA1 Message Digest from SUN, <initialized>

SHA1("") =DA39A3EE5E6B4B0D3255BFEF95601890AFD80709

SHA1("abc") = A9993E364706816ABA3E25717850C26C9CD0D89D

SHA1("abcdefghijklmnopqrstuvwxyz") = 32D10C7B8CF96570CA04CE37F2A19D84240D3A89

11. Calculate the message digest of a text using the MD5 algorithm in JAVA.

The MD5 message-digest algorithm is a widely used cryptographic hash function producing a 128-bit (16-byte) hash value, typically expressed in text format as a 32-digit hexadecimal number. MD5 has been utilized in a wide variety of cryptographic applications and is also commonly used to verify data integrity

MD5 processes a variable-length message into a fixed length output of 128 bits. The input message is broken up into chunks of 512 bit blocks. The message is padded so that its length is divisible by 512. The padding works as follows: first a single bit, 1, is appended to the end of the message. This is followed by as many zeros as are required to bring the length of the message up to 64 bits less than a multiple of 512. The remaining bits are filled up with 64 bits representing the length of the original message, modulo 2^{64}. The main MD5 algorithm operates on a 128-bit state, divided into four 32 – bit words, denoted A, B, C and D. These are initialized to certain fixed constants. The main algorithm then uses each 512-bit message block in turn to modify the state.

ALGORITHM:

- Read the 128-bit plain text.
- Divide into four blocks of 32-bits named as A, B, C and D.
- Compute the functions f, g, h and i with operations such as, rotations, permutations, etc,.
- The output of these functions are combined together as F and performed circular shifting and then given to key round.
- Finally, right shift of ‘s’ times are performed and the results are combined together to produce the final output.

Expected Output:

Message digest object info:
Algorithm = MD5
Provider = SUN version 1.8
ToString = MD5 Message Digest from SUN, <initialized>

MD5("") = D41D8CD98F00B204E9800998ECF8427E

MD5("abc") = 900150983CD24FB0D6963F7D28E17F72

MD5("abcdefghijklmnopqrstuvwxyz") = C3FCD3D76192E4007DFB496CCA67E13B