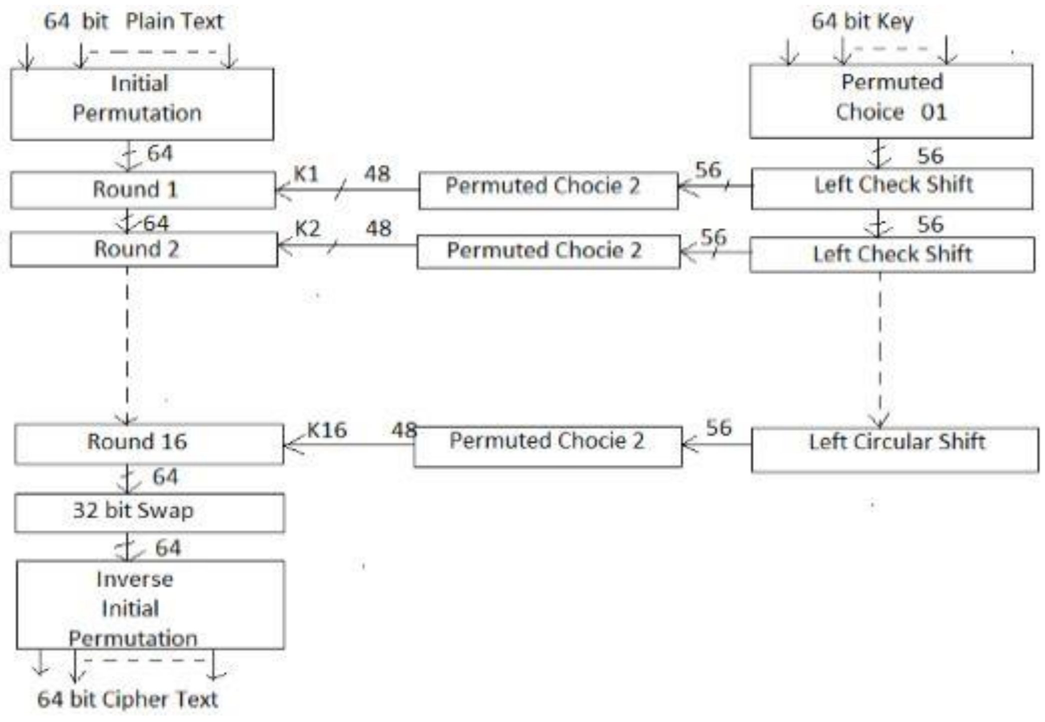
Exercise 1: Write a JAVA program to implement the DES algorithm logic.

Digital Encryption Standard (DES) Algorithm

- It is the most widely used algorithm for encryption and it is known as data encryption algorithm. (DEA)
- In DES, data are encrypted in 64 blocks bits using a 56 bit key.
- The algorithm transforms the 64 bit input in a series of steps into a 64 bit output using 56 bit key.
- The same steps are used to reverse encryption using the same key.
- The overall scheme of DES encryption is illustrated in the below figure.



Overall Scheme of DES algorithm

1. There are 2 inputs to the encryption function, the plain text of 64 bits (to be encrypted)
2. The key of 64 bits (actually 56 bits + 8 parity bits)
3. The processing of plaintext (LHS) of above figure proceeds in three phases.
   i. First the 64 bit plaintext passes through an initial permutation that rearranges the bits to produce permitted output.

ii. Output of last round consists of 64 bits that are a function of input plain text and key. The left and right halves of input are swapped to produce pre-output.

iii. Finally the pre output is passed through inverse permutation (initially used) to produce 64 bit cipher text.

4. The R.H.S of figure shows the way in which 56 bit key is used.
5. Then for each round, a subkey R_i is produced by combination of left circular shift and permutation.
6. The permutation function is same for each round but a different subkey is produced because of repeated shift of key bits.

Single Round of DES algorithm

- The following figure shows the internal structure of a single round.
- The left and right halves of each 64 bit intermediate value are treated as separate 32 bit quantities labeled left (L) and Right (R)

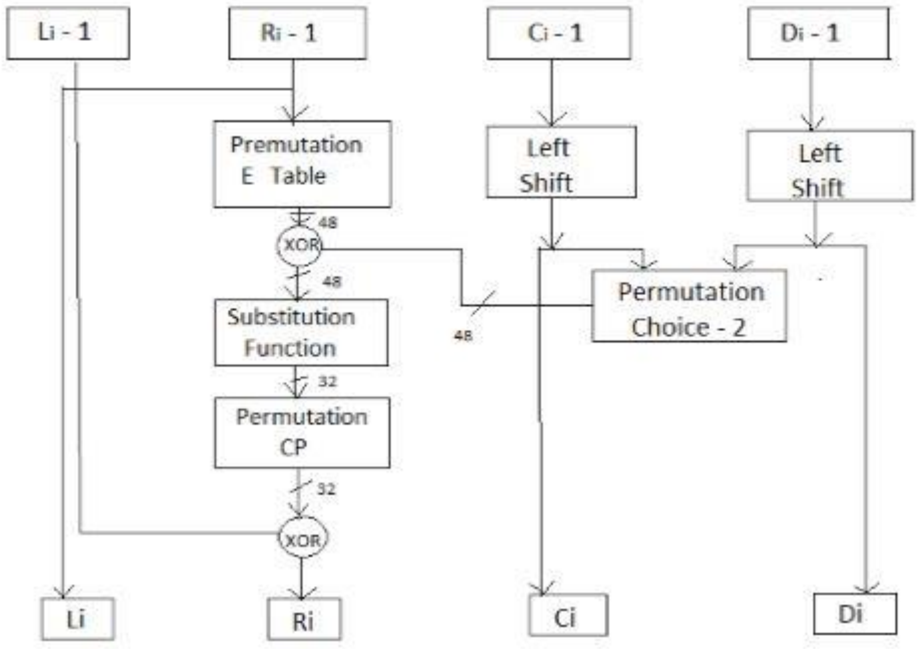


Figure 4.8 Single Round of DES algorithm

- The overall processing of each channel summarized as:

$L_i = R_i - 1$
$R_i = L_i - 1 \text{ XOR } F(R_i - 1 , k_i)$

Example: Let **M** be the plain text message **M** = 0123456789ABCDEF, where **M** is in hexadecimal (base 16) format. Rewriting **M** in binary format, we get the 64-bit block of text:

M = 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
L = 0000 0001 0010 0011 0100 0101 0110 0111
R = 1000 1001 1010 1011 1100 1101 1110 1111

The first bit of **M** is "0". The last bit is "1". We read from left to right.

DES operates on the 64-bit blocks using *key* sizes of 56- bits. The keys are actually stored as being 64 bits long, but every 8th bit in the key is not used (i.e. bits numbered 8, 16, 24, 32, 40, 48, 56, and 64). However, we will nevertheless number the bits from 1 to 64, going left to right, in the following calculations. But, as you will see, the eight bits just mentioned get eliminated when we create subkeys.

Example: Let **K** be the hexadecimal key **K** = 133457799BBCDFF1. This gives us as the binary key (setting 1 = 0001, 3 = 0011, etc., and grouping together every eight bits, of which the last one in each group will be unused):

K = 00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001

The DES algorithm uses the following steps:

Step 1: Create 16 subkeys, each of which is 48-bits long.

The 64-bit key is permuted according to the following table, **PC-1**. Since the first entry in the table is "57", this means that the 57th bit of the original key **K** becomes the first bit of the permuted key **K**+. The 49th bit of the original key becomes the second bit of the permuted key. The 4th bit of the original key is the last bit of the permuted key. Note only 56 bits of the original key appear in the permuted key.

PC-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

Example: From the original 64-bit key

K = 00010011 00110100 01010111 01111001 10011011 10111100 11011111 11110001

we get the 56-bit permutation

K+ = 1111000 0110011 0010101 0101111 0101010 1011001 1001111 0001111

Next, split this key into left and right halves, C_0 and D_0, where each half has 28 bits.

Example: From the permuted key **K**+, we get

C_0 = 1111000 0110011 0010101 0101111
D_0 = 0101010 1011001 1001111 0001111

With C_0 and D_0 defined, we now create sixteen blocks C_n and D_n, 1<=n<=16. Each pair of blocks C_n and D_n is formed from the previous pair C_{n-1} and D_{n-1}, respectively, for n = 1, 2, ..., 16, using the following schedule of "left shifts" of the previous block. To do a left shift, move each bit one place to the left, except for the first bit, which is cycled to the end of the block.

Iteration Number	Number of Left Shifts
1	1
2	1

3	2
4	2
5	2
6	2
7	2
8	2
9	1
10	2
11	2
12	2
13	2
14	2
15	2
16	1

This means, for example, C_3 and D_3 are obtained from C_2 and D_2, respectively, by two left shifts, and C_{16} and D_{16} are obtained from C_{15} and D_{15}, respectively, by one left shift. In all cases, by a single left shift is meant a rotation of the bits one place to the left, so that after one left shift the bits in the 28 positions are the bits that were previously in positions 2, 3,..., 28, 1.

Example: From original pair pair C_0 and D_0 we obtain:

C_0 = 1111000011001100101010101111
D_0 = 0101010101100110011110001111

C_1 = 1110000110011001010101011111
D_1 = 1010101011001100111100011110

C_2 = 1100001100110010101010111111
D_2 = 0101010110011001111000111101

C_3 = 0000110011001010101011111111
D_3 = 0101011001100111100011110101

C_4 = 0011001100101010101111111100

D_4 = 0101100110011110001111010101

C_5 = 1100110010101010111111110000

D_5 = 0110011001111000111101010101

C_6 = 0011001010101011111111000011

D_6 = 1001100111100011110101010101

C_7 = 1100101010101111111100001100

D_7 = 0110011110001111010101010110

C_8 = 0010101010111111110000110011

D_8 = 1001111000111101010101011001

C_9 = 0101010101111111100001100110

D_9 = 0011110001111010101010110011

C_{10} = 0101010111111110000110011001

D_{10} = 1111000111101010101011001100

C_{11} = 0101011111111000011001100101

D_{11} = 1100011110101010101100110011

C_{12} = 0101111111100001100110010101

D_{12} = 0001111010101010110011001111

C_{13} = 0111111110000110011001010101

D_{13} = 0111101010101011001100111100

C_{14} = 1111111000011001100101010101

D_{14} = 1110101010101100110011110001

C_{15} = 1111100001100110010101010111

D_{15} = 1010101010110011001111000111

C_{16} = 1111000011001100101010101111
D_{16} = 0101010101100110011110001111

We now form the keys ***K_n***, for 1<=***n***<=16, by applying the following permutation table to each of the concatenated pairs ***C_nD_n***. Each pair has 56 bits, but **PC-2** only uses 48 of these.

PC-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

Therefore, the first bit of ***K_n*** is the 14th bit of ***C_nD_n***, the second bit the 17th, and so on, ending with the 48th bit of ***K_n*** being the 32th bit of ***C_nD_n***.

Example: For the first key we have ***C_1D_1*** = 1110000 1100110 0101010 1011111 1010101 0110011 0011110 0011110

which, after we apply the permutation **PC-2**, becomes

K_1 = 000110 110000 001011 101111 111111 000111 000001 110010

For the other keys we have

K_2 = 011110 011010 111011 011001 110110 111100 100111 100101
K_3 = 010101 011111 110010 001010 010000 101100 111110 011001
K_4 = 011100 101010 110111 010110 110110 110011 010100 011101
K_5 = 011111 001110 110000 000111 111010 110101 001110 101000
K_6 = 011000 111010 010100 111110 010100 000111 101100 101111
K_7 = 111011 001000 010010 110111 111101 100001 100010 111100

K_8 = 111101 111000 101000 111010 110000 010011 101111 111011

K_9 = 111000 001101 101111 101011 111011 011110 011110 000001

K_{10} = 101100 011111 001101 000111 101110 100100 011001 001111

K_{11} = 001000 010101 111111 010011 110111 101101 001110 000110

K_{12} = 011101 010111 000111 110101 100101 000110 011111 101001

K_{13} = 100101 111100 010111 010001 111110 101011 101001 000001

K_{14} = 010111 110100 001110 110111 111100 101110 011100 111010

K_{15} = 101111 111001 000110 001101 001111 010011 111100 001010

K_{16} = 110010 110011 110110 001011 000011 100001 011111 110101

So much for the subkeys. Now we look at the message itself.

Step 2: Encode each 64-bit block of data.

There is an *initial permutation* **IP** of the 64 bits of the message data **M**. This rearranges the bits according to the following table, where the entries in the table show the new arrangement of the bits from their initial order. The 58th bit of **M** becomes the first bit of **IP**. The 50th bit of **M** becomes the second bit of **IP**. The 7th bit of**M** is the last bit of **IP**.

IP

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

Example: Applying the initial permutation to the block of text **M**, given previously, we get

M = 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111
IP = 1100 1100 0000 0000 1100 1100 1111 1111 1111 0000 1010 1010 1111 0000 1010 1010

Here the 58th bit of **M** is "1", which becomes the first bit of **IP**. The 50th bit of **M** is "1", which becomes the second bit of **IP**. The 7th bit of **M** is "0", which becomes the last bit of **IP**.

Next divide the permuted block **IP** into a left half L_0 of 32 bits, and a right half R_0 of 32 bits.

Example: From **IP**, we get L_0 and R_0

L_0 = 1100 1100 0000 0000 1100 1100 1111 1111
R_0 = 1111 0000 1010 1010 1111 0000 1010 1010

We now proceed through 16 iterations, for $1 <= n <= 16$, using a function f which operates on two blocks--a data block of 32 bits and a key K_n of 48 bits--to produce a block of 32 bits. **Let + denote XOR addition, (bit-by-bit addition modulo 2)**. Then for **n** going from 1 to 16 we calculate

$L_n = R_{n-1}$
$R_n = L_{n-1} + f(R_{n-1}, K_n)$

This results in a final block, for $n = 16$, of $L_{16}R_{16}$. That is, in each iteration, we take the right 32 bits of the previous result and make them the left 32 bits of the current step. For the right 32 bits in the current step, we XOR the left 32 bits of the previous step with the calculation f .

Example: For $n = 1$, we have

K_1 = 000110 110000 001011 101111 111111 000111 000001 110010
$L_1 = R_0$ = 1111 0000 1010 1010 1111 0000 1010 1010
$R_1 = L_0 + f(R_0, K_1)$

It remains to explain how the function ***f*** works. To calculate ***f***, we first expand each block $\boldsymbol{R_{n-1}}$ from 32 bits to 48 bits. This is done by using a selection table that repeats some of the bits in $\boldsymbol{R_{n-1}}$. We'll call the use of this selection table the function **E**. Thus **E**($\boldsymbol{R_{n-1}}$) has a 32 bit input block, and a 48 bit output block.

Let **E** be such that the 48 bits of its output, written as 8 blocks of 6 bits each, are obtained by selecting the bits in its inputs in order according to the following table:

E BIT-SELECTION TABLE

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

Thus the first three bits of **E**($\boldsymbol{R_{n-1}}$) are the bits in positions 32, 1 and 2 of $\boldsymbol{R_{n-1}}$ while the last 2 bits of **E**($\boldsymbol{R_{n-1}}$) are the bits in positions 32 and 1.

Example: We calculate **E**($\boldsymbol{R_0}$) from $\boldsymbol{R_0}$ as follows:

$\boldsymbol{R_0}$ = 1111 0000 1010 1010 1111 0000 1010 1010
E($\boldsymbol{R_0}$) = 011110 100001 010101 010101 011110 100001 010101 010101

(Note that each block of 4 original bits has been expanded to a block of 6 output bits.)

Next in the ***f*** calculation, we XOR the output **E**($\boldsymbol{R_{n-1}}$) with the key $\boldsymbol{K_n}$:

$$\boldsymbol{K_n} + \mathbf{E}(\boldsymbol{R_{n-1}}).$$

Example: For $\boldsymbol{K_1}$, **E**($\boldsymbol{R_0}$), we have

$\boldsymbol{K_1}$ = 000110 110000 001011 101111 111111 000111 000001 110010

E($\boldsymbol{R_0}$) = 011110 100001 010101 010101 011110 100001 010101 010101

$\boldsymbol{K_1}$+**E**($\boldsymbol{R_0}$) = 011000 010001 011110 111010 100001 100110 010100 100111.

We have not yet finished calculating the function ***f*** . To this point we have expanded $\boldsymbol{R_{n-1}}$ from 32 bits to 48 bits, using the selection table, and XORed the result with the key $\boldsymbol{K_n}$. We now have 48 bits, or eight groups of six bits. We now do something strange with each group of six bits: we use them as addresses in tables called "**S boxes**". Each group of six bits will give us an address in a different **S** box. Located at that address will be a 4 bit number. This 4 bit number will replace the original 6 bits. The net result is that the eight groups of 6 bits are transformed into eight groups of 4 bits (the 4-bit outputs from the **S** boxes) for 32 bits total.

Write the previous result, which is 48 bits, in the form:

$$\boldsymbol{K_n} + \mathbf{E}(\boldsymbol{R_{n-1}}) = \boldsymbol{B_1B_2B_3B_4B_5B_6B_7B_8},$$

where each $\boldsymbol{B_i}$ is a group of six bits. We now calculate

$$\boldsymbol{S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8)}$$

where $\boldsymbol{S_i(B_i)}$ referres to the output of the ***i***-th **S** box.

To repeat, each of the functions ***S1, S2,..., S8***, takes a 6-bit block as input and yields a 4-bit block as output. The table to determine $\boldsymbol{S_1}$ is shown and explained below:

S1

	Column Number															
Row No.	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
0	**14**	**4**	**13**	**1**	**2**	**15**	**11**	**8**	**3**	**10**	**6**	**12**	**5**	**9**	**0**	**7**
1	**0**	**15**	**7**	**4**	**14**	**2**	**13**	**1**	**10**	**6**	**12**	**11**	**9**	**5**	**3**	**8**
2	**4**	**1**	**14**	**8**	**13**	**6**	**2**	**11**	**15**	**12**	**9**	**7**	**3**	**10**	**5**	**0**
3	**15**	**12**	**8**	**2**	**4**	**9**	**1**	**7**	**5**	**11**	**3**	**14**	**10**	**0**	**6**	**13**

If $\boldsymbol{S_1}$ is the function defined in this table and ***B*** is a block of 6 bits, then $\boldsymbol{S_1(B)}$ is determined as follows: The first and last bits of ***B*** represent in base 2 a number in the decimal range 0 to 3 (or binary 00 to 11). Let that number be ***i***. The middle 4 bits of ***B*** represent in base 2 a number in the decimal range 0 to 15 (binary 0000 to 1111). Let that number be ***j***. Look up in the table the number in the ***i***-th row and ***j***-th column. It is a number in the range 0 to 15 and is uniquely represented by a 4 bit block. That block is the output $\boldsymbol{S_1(B)}$ of $\boldsymbol{S_1}$ for the input ***B***. For example, for input block ***B*** = 011011 the first bit is "0" and the last bit "1" giving 01 as the row. This is row 1. The middle four bits are "1101". This is the binary equivalent of decimal 13, so the column is column number 13. In row 1, column 13 appears 5. This determines the output; 5 is binary 0101, so that the output is 0101. Hence $\boldsymbol{S_1}$(011011) = 0101.

The tables defining the functions $\boldsymbol{S_1,\ldots,S_8}$ are the following:

S1

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

S2

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

S3

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

S4

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

S5

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

S6

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

S7

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

S8

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

Example: For the first round, we obtain as the output of the eight **S** boxes:

K_1 + **E**(R_0) = 011000 010001 011110 111010 100001 100110 010100 100111.

$S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8)$ = 0101 1100 1000 0010 1011 0101 1001 0111

The final stage in the calculation of ***f*** is to do a permutation **P** of the **S**-box output to obtain the final value of ***f***:

$$f = \mathbf{P}(S_1(B_1)S_2(B_2)...S_8(B_8))$$

The permutation **P** is defined in the following table. **P** yields a 32-bit output from a 32-bit input by permuting the bits of the input block.

P

16	7	20	21
29	12	28	17
1	15	23	26
5	18	31	10
2	8	24	14
32	27	3	9
19	13	30	6
22	11	4	25

Example: From the output of the eight **S** boxes:

$S_1(B_1)S_2(B_2)S_3(B_3)S_4(B_4)S_5(B_5)S_6(B_6)S_7(B_7)S_8(B_8)$ = 0101 1100 1000 0010 1011 0101 1001 0111

we get

$$f = 0010\ 0011\ 0100\ 1010\ 1010\ 1001\ 1011\ 1011$$

$R_1 = L_0 + f(R_0 , K_1)$

= 1100 1100 0000 0000 1100 1100 1111 1111
+ 0010 0011 0100 1010 1010 1001 1011 1011
= 1110 1111 0100 1010 0110 0101 0100 0100

In the next round, we will have $\boldsymbol{L_2} = \boldsymbol{R_1}$, which is the block we just calculated, and then we must calculate $\boldsymbol{R_2 = L_1 + f(R_1, K_2)}$, and so on for 16 rounds. At the end of the sixteenth round we have the blocks $\boldsymbol{L_{16}}$ and $\boldsymbol{R_{16}}$. We then ***reverse*** the order of the two blocks into the 64-bit block

$$\boldsymbol{R_{16}L_{16}}$$

and apply a final permutation **IP^{-1}** as defined by the following table:

IP^{-1}

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

That is, the output of the algorithm has bit 40 of the preoutput block as its first bit, bit 8 as its second bit, and so on, until bit 25 of the preoutput block is the last bit of the output.

Example: If we process all 16 blocks using the method defined previously, we get, on the 16th round,

$\boldsymbol{L_{16}}$ = 0100 0011 0100 0010 0011 0010 0011 0100
$\boldsymbol{R_{16}}$ = 0000 1010 0100 1100 1101 1001 1001 0101

We reverse the order of these two blocks and apply the final permutation to

$R_{16}L_{16}$ = 00001010 01001100 11011001 10010101 01000011 01000010 00110010 00110100

IP^{-1} = 10000101 11101000 00010011 01010100 00001111 00001010 10110100 00000101

which in hexadecimal format is

85E813540F0AB405.

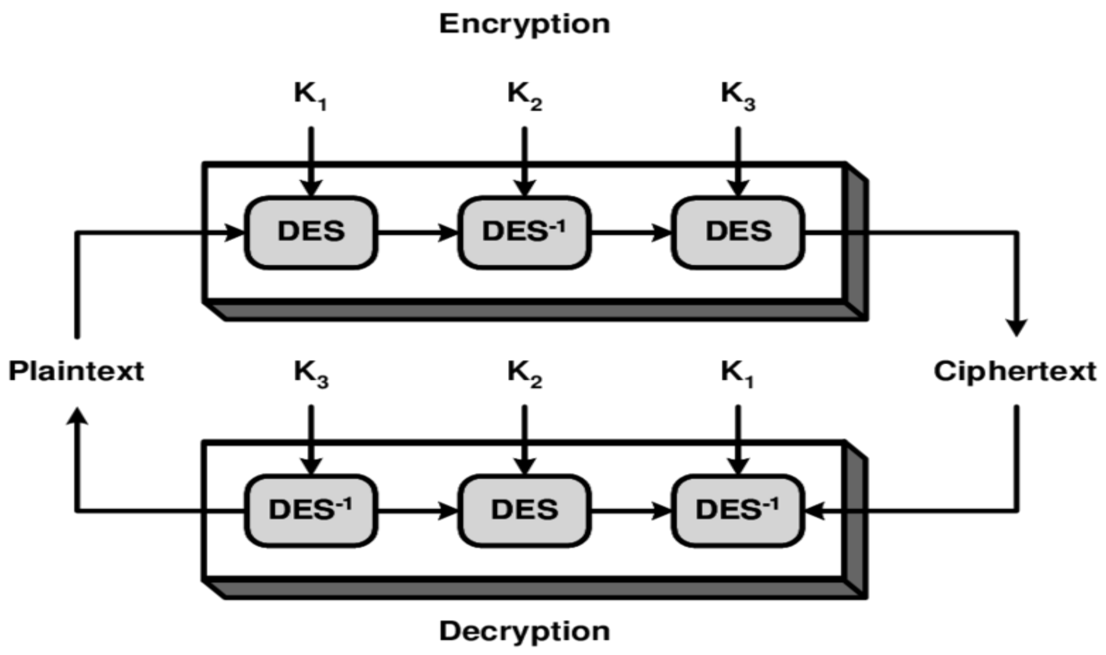
This is the encrypted form of **M** = 0123456789ABCDEF: namely, **C** = 85E813540F0AB405.

Decryption is simply the inverse of encryption, following the same steps as above, but reversing the order in which the subkeys are applied.

Exercise 2: Write a JAVA program that contains functions, which accept a key and input text to be encrypted / decrypted. This program should use the key to encrypt/decrypt the input using the triple DES algorithm. Make use of Java Cryptography Package.

- Triple-DES is just DES with two 56-bit keys applied.
- Given a plaintext message,
  - the first key is used to DES- encrypt the message.
  - the second key is used to DES-decrypt the encrypted message. (Since the second key is not the right key, this decryption just scrambles the data further.)
- The twice-scrambled message is then encrypted again with the first key to yield the final ciphertext.
- This three-step procedure is called triple-DES.

Triple-DES is just DES done three times with two keys used in a particular order. (Triple-DES can also be done with three separate keys instead of only two. In either case the resultant key space is about 2^112.)

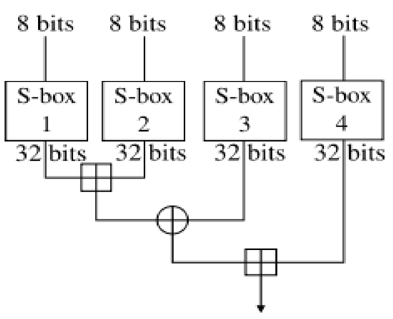


Exercise 3: Write a JAVA program to implement the Blowfish algorithm.

The Blowfish algorithm

- Blowfish has a 64-bit block size and a key length of anywhere from 32 bits to 448 bits.

- It is a 16-round Feistel cipher and uses large key-dependent S-boxes.



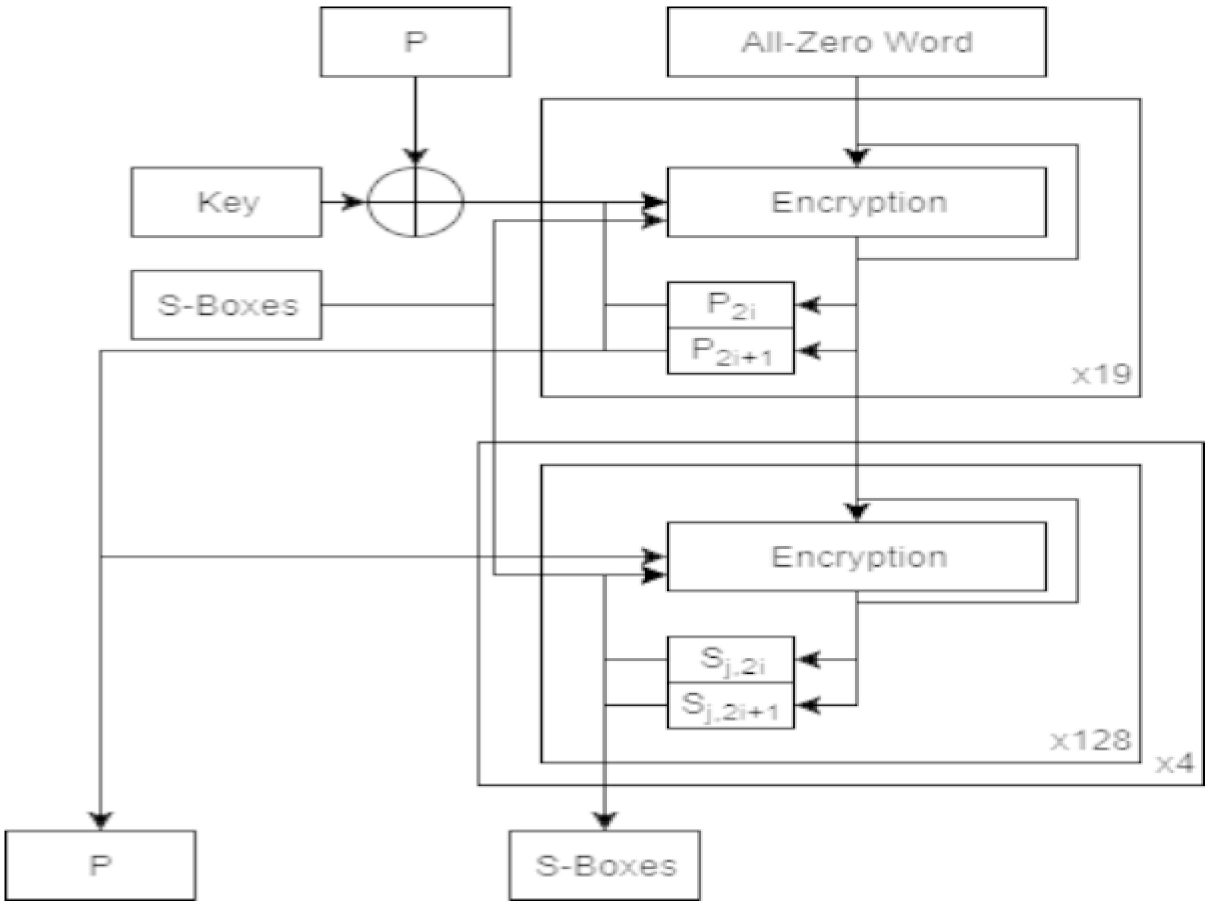
The Feistel structure of Blowfish

- The diagram to the left shows the action of Blowfish.
  - Each line represents 32 bits.
  - The algorithm keeps two sub key arrays: the 18-entry P-array and four 256-entry S-boxes.
  - The S-boxes accept 8-bit input and produce 32-bit output.
  - One entry of the P-array is used every round, and after the final round, each half of the data block is XORed with one of the two remaining unused P-entries.

- The diagram to the right shows Blowfish's F-function.
  - The function splits the 32-bit input into four eight-bit quarters, and uses the quarters as input to the S-boxes.
  - The outputs are added modulo 232 and XORed to produce the final 32-bit output.
  - Since Blowfish is a Feistel network, it can be inverted simply by XORing P17 and P18 to the ciphertext block, then using the P-entries in reverse order.

Steps of Blow Fish Algorithm

1. Blowfish's key schedule starts by initializing the P-array and S-boxes with values derived from the hexadecimal digits of pi, which contain no obvious pattern.
2. The secret key is then XORed with the P-entries in order (cycling the key if necessary).
3. A 64-bit all-zero block is then encrypted with the algorithm as it stands.
4. The resultant cipher text replaces P1 and P2.
5. The cipher text is then encrypted again with the new subkeys, and P3 and P4 are replaced by the new cipher text. This continues, replacing the entire P-array and all the S-box entries.
6. In all, the Blowfish encryption algorithm will run 521 times to generate all the subkeys - about 4KB of data is processed.



Exercise-4 : Using Java Cryptography, encrypt the text “Hello world”, using Blowfish. Create your own tool using Java Key tool.

Program:

```
import javax.crypto.Cipher;

import javax.crypto.KeyGenerator;

import javax.crypto.SecretKey;

import javax.swing.JOptionPane;

public class BlowFishCipher

{

        public static void main(String[] args) throws Exception

        {

                // create a key generator based upon the Blowfish cipher

                KeyGenerator keygenerator = KeyGenerator.getInstance("Blowfish");

                // create a key

                SecretKey secretkey = keygenerator.generateKey();

                // create a cipher based upon Blowfish

                Cipher cipher = Cipher.getInstance("Blowfish");

                // initialise cipher to with secret key

                cipher.init(Cipher.ENCRYPT_MODE, secretkey);

                // get the text to encrypt

                String inputText = JOptionPane.showInputDialog("Input your message:
");

                // encrypt message
                byte[] encrypted = cipher.doFinal(inputText.getBytes());

                // re-initialise the cipher to be in decrypt mode

                cipher.init(Cipher.DECRYPT_MODE, secretkey);

                // decrypt message

                byte[] decrypted = cipher.doFinal(encrypted);

                // and display the results

                JOptionPane.showMessageDialog(JOptionPane.getRootFrame(),

                "\nEncrypted text: " + new String(encrypted) + "\n" +

                "\nDecrypted text: " + new String(decrypted));

                System.exit(0);

        }

}
```

Output:

```
Input your message: Hello world
Encrypted text: 3ooo&&(*&*4r4
Decrypted text: Hello world
```

Exercise 5: Write a Java Program to implement RSA algorithm

RSA Public-key encryption algorithm

- RSA involves a public-key and a private-key where the public key is known to all and is used to encrypt data or message. The data or message which has been encrypted using a public key can only be decrypted by using its corresponding private-key.
- RSA is a block cipher in which plain text and cipher text are integers between 0 and n-1 for some n.

Key Generation

- Select p, q
  - such that (i) p and q are both prime;
    (ii) $p \neq q$
- Calculate $n = p \times q$
  $Ø(n) = (p-1)(q-1)$
- Select integer e
  - such that (i) $\gcd(Ø(n),e) = 1$;
    (ii) $1<e< Ø(n)$
- Calculate d
  $de \bmod Ø(n)=1$

Encryption and Decryption are of the following form:

For some plain text block M and cipher text block C

$C = M^e \bmod n$ (Encryption)

$M = C^d \bmod n$ (Decryption)

- Both sender and receiver must know the values of **n** and **e** and only the receiver knows the value of **d**

Public Key KU = {e,n}

Private Key KR = {d,n}

Example:

1. **Generating Public Key**
   - Select two prime no's **P=17** and **Q=11**
   - Compute **n = P*Q = 187**
   - Compute **Φ(n) = (P-1) (Q-1)= 160**
   - Select **e**
     - ➢ **gcd(e,160) = 1**
     - ➢ choose **e = 7**
   - Our public key is made of **e** and **n : 7, 187**

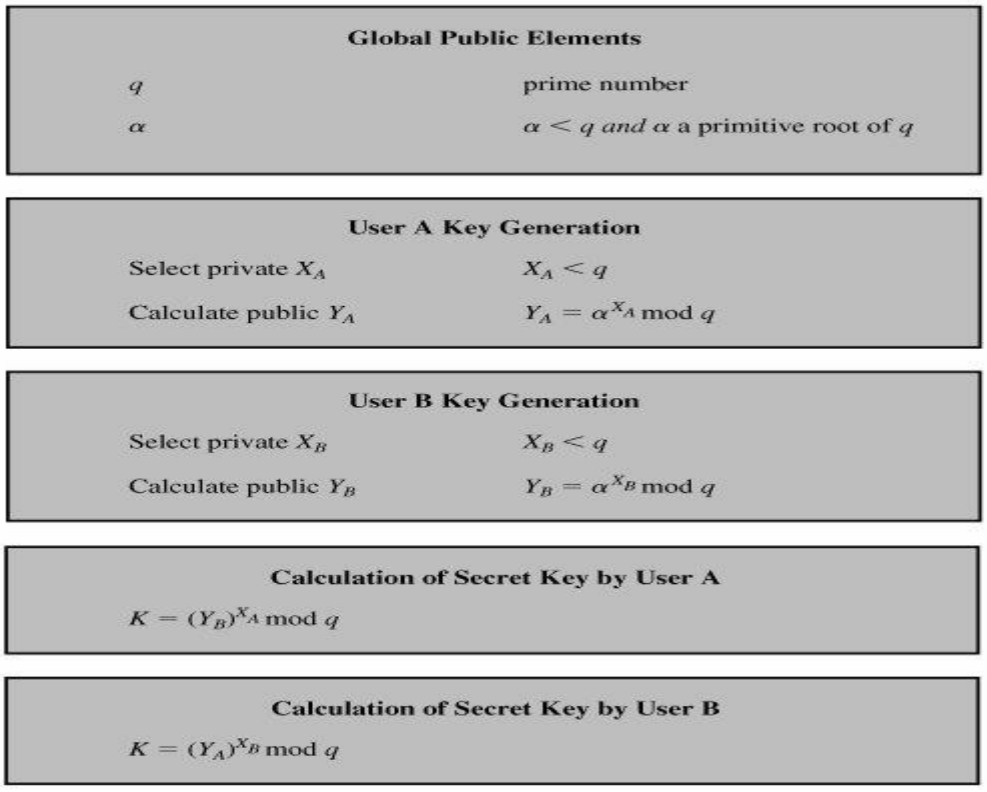
2. **Generating Private Key**
   - Determine **d**
     - ➢ **de = 1 mod 160** and **d <160**
     - ➢ **d = 23** *since 23 x 7* **=** *161* **=** *10 x 160 + 1*
   - Out private key is made of **d** and **n: 23, 187**

Now, given message M = 88

3. **Encryption**
   - Encryption $C = M^e \bmod n$
   - $88^7 \bmod 187 = 11$
4. **Decryption**
   - Decryption $M = C^d \bmod n$
   - $M = 11^{23} \bmod 187 = 88$

Exercise 6: Implement the Diffie-Hellman Key Exchange mechanism using HTML and JavaScript. Consider the end user as one of the part(Alice) and the JavaScript application as the other party(Bob)

Diffie-Hellman key exchange (**D-H**) is a cryptographic protocol that enables two users to exchange a secret securely that can be used for subsequent encryption of messages. The algorithm itself is limited to the exchange of the keys.This key can then be used to encrypt subsequent communications using a symmetric key cipher.



Global Public Elements

q	prime number
α	$\alpha < q$ *and* α a primitive root of q

User A Key Generation

Select private X_A	$X_A < q$
Calculate public Y_A	$Y_A = \alpha^{X_A} \bmod q$

User B Key Generation

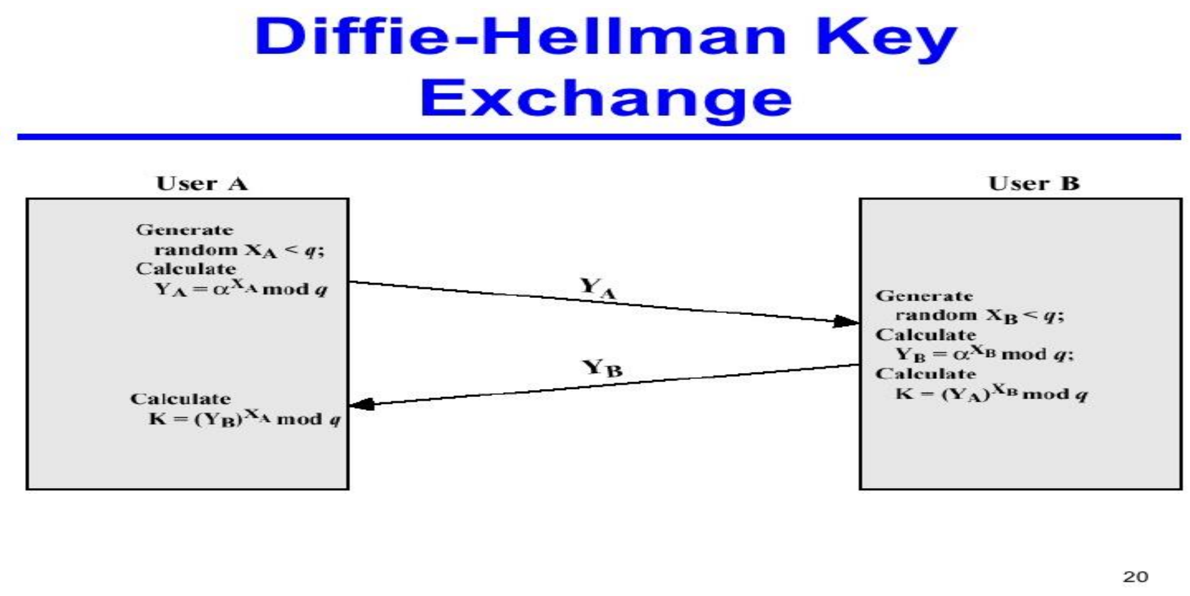
Select private X_B	$X_B < q$
Calculate public Y_B	$Y_B = \alpha^{X_B} \bmod q$

Calculation of Secret Key by User A

$K = (Y_B)^{X_A} \bmod q$

Calculation of Secret Key by User B

$K = (Y_A)^{X_B} \bmod q$


Diffie-Hellman Key Exchange
User A
User B
Generate random $X_A < q$; Calculate $Y_A = \alpha^{X_A} \bmod q$
Calculate $K = (Y_B)^{X_A} \bmod q$
Y_A
Y_B
Generate random $X_B < q$; Calculate $Y_B = \alpha^{X_B} \bmod q$; Calculate $K = (Y_A)^{X_B} \bmod q$
20

- For this scheme, there are two publicly known numbers:
  - a prime number **q** and an integer **α** that is a primitive root of q.
- Suppose the users A and B wish to exchange a key.
  - User A selects a random integer $\mathbf{X_A < q}$ and computes $\mathbf{Y_A = \alpha X_A \bmod q}$.
  - Similarly, user B independently selects a random integer $\mathbf{X_A < q}$ and computes $\mathbf{Y_B = \alpha X_B \bmod q.}$
  - Each side keeps the **X** value private and makes the **Y** value available publicly to the other side.
  - User A computes the key as $\mathbf{K = (Y_B)X_A \bmod q}$
  - User B computes the key as $\mathbf{K = (Y_A)X_B \bmod q}$. These two calculations produce identical results.

Example

Step 1: Alice and Bob get public numbers P=23, G=9

Step 2: Alice selected a private key a = 4 and

Bob selected a private key b = 3

Step 3: Alice and Bob computer public values

Alice: x = (9 ^ 4 mod 23) = (6561 mod 23) = **6**

Bob : y = (9 ^ 3 mod 23) = (729 mod 23) = **16**

Step 4: Alice and Bob exchange public numbers

Step 5: Alice received public key y = 16 and

Bob receives public key x = 6

Step 6: Alice and Bob compute symmetric keys

Alice k_a = y ^ a mod p = 65536 mod 23 = 9

Bob k_b = x ^ b mod p = 216 mod 23 = 9

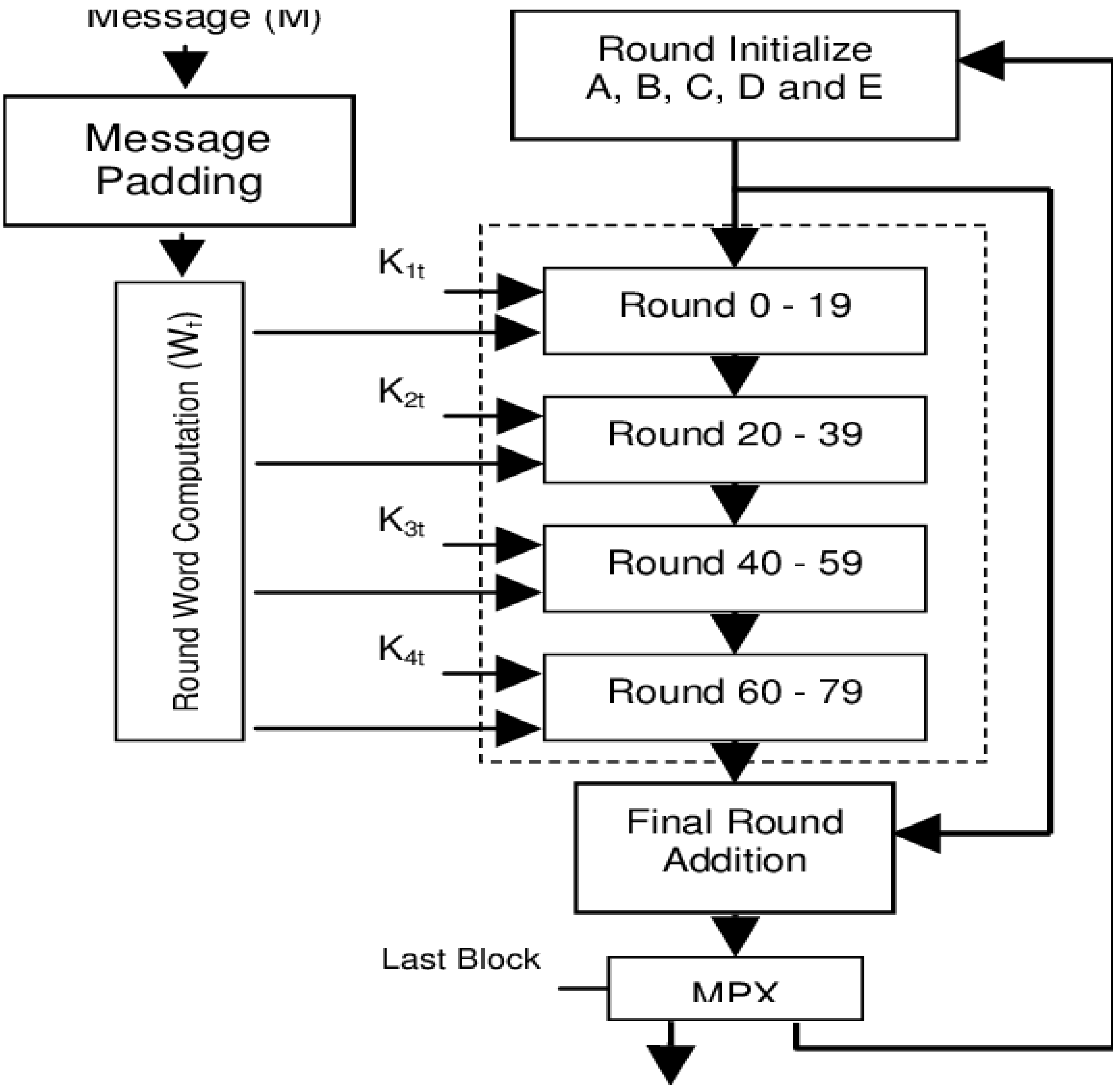
Step 7: 9 is the shared secret.

Exercise 7: Calculate the message digest of a text using the SHA-1 algorithm in JAVA

Secure Hash Algorithm 1 (SHA-1)

SHA1 (Secure Hash Algorithm-1) is message-digest algorithm, which takes an input message of any length < 2^64 bits and produces a 160-bit output as the message digest.

Based on the SHA1 RFC document, the SHA-1 is called secure because it is computationally infeasible to find a message which corresponds to a given message digest, or to find two different messages which produce the same message digest. Any change to a message in transit will, with very high probability, result in a different message digest, and the signature will fail to verify.



SHA1 algorithm consists of 6 tasks:

Task 1. Appending Padding Bits. The original message is "padded" (extended) so that its length (in bits) is congruent to 448, modulo 512. The padding rules are:

- The original message is always padded with one bit "1" first.
- Then zero or more bits "0" are padded to bring the length of the message up to 64 bits less than a multiple of 512.

Task 2. Appending Length. 64 bits are appended to the end of the padded message to indicate the length of the original message in bytes. The rules of appending length are:

- The length of the original message in bytes is converted to its binary format of 64 bits. If overflow happens, only the low-order 64 bits are used.
- Break the 64-bit length into 2 words (32 bits each).
- The low-order word is appended first and followed by the high-order word.

Task 3. Preparing 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)

Task 4. Preparing 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)

Task 5. Initializing Buffers. SHA1 algorithm requires 5 word buffers with the following initial values:

H0 = 0x67452301

H1 = 0xEFCDAB89

H2 = 0x98BADCFE

H3 = 0x10325476

H4 = 0xC3D2E1F0

Task 6. Processing Message in 512-bit Blocks. This is the main task of SHA1 algorithm, which loops through the padded and appended message in blocks of 512 bits each. For each input block, a number of operations are performed. This task can be described in the following pseudo code slightly modified from the RFC 3174's method 1:

Input and predefined functions:

M[1, 2, ..., N]: Blocks of the padded and appended message

f(0;B,C,D), f(1,B,C,D), ..., f(79,B,C,D): Defined as above

K(0), K(1), ..., K(79): Defined as above

H0, H1, H2, H3, H4: Word buffers with initial values

Algorithm:

```
For loop on k = 1 to N
  (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
```

Output:

H0, H1, H2, H3, H4: Word buffers with final message digest

Exercise 8: Calculate the message digest of a text using the MD5 algorithm in Java

Preparing the input

- The MD5 algorithm first divides the input in **blocks** of 512 bits each.
- 64 Bits are inserted at the end of the last block.
- These 64 bits are used to record the length of the original input. If the last block is less than 512 bits, some extra bits are 'padded' to the end.
- Next, each **block** is divided into 16 **words** of 32 bits each. These are denoted as M_0 ... M_{15}.

MD5 helper functions

The buffer

MD5 uses a buffer that is made up of four **words** that are each 32 bits long. These words are called A, B, C and D. They are initialized as

word A: 01 23 45 67

word B: 89 ab cd ef

word C: fe dc ba 98

word D: 76 54 32 10

The table

MD5 further uses a table K that has 64 elements. Element number i is indicated as K_i. The table is computed beforehand to speed up the computations. The elements are computed using the mathematical sin function:

$K_i = abs(sin(i + 1)) * 2^{32}$

Four auxiliary functions

In addition MD5 uses four auxiliary functions that each take as input three 32-bit words and produce as output one 32-bit word. They apply the logical operators and, or, not and xor to the input bits.

F(X,Y,Z) = (X and Y) or (not(X) and Z)

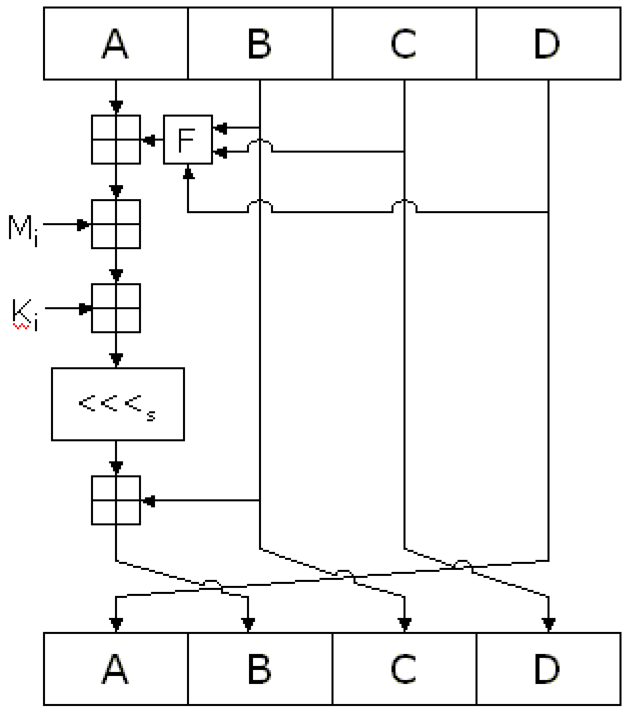
G(X,Y,Z) = (X and Z) or (Y and not(Z))

H(X,Y,Z) = X xor Y xor Z

I(X,Y,Z) = Y xor (X or not(Z))

Processing the blocks

The contents of the four buffers (A, B, C and D) are now mixed with the words of the input, using the four auxiliary functions (F, G, H and I). There are four *rounds*, each involves 16 basic *operations*. One operation is illustrated in the figure below.



The figure shows how the auxiliary function F is applied to the four buffers (A, B, C and D), using message word M_i and constant K_i. The item "<<<s" denotes a binary left shift by *s* bits.

The output

After all rounds have been performed, the buffers A, B, C and D contain the MD5 digest of the original input.

Exercise 9: Explore the Java classes related to digital certificates.

For the purposes of digital signing of documents, verification of digital signatures, and handling digital certificates in the Java platform, the Java Cryptography Architecture (JCA) is used. JCA is a specification that gives the programmers a standard way to access cryptographic services, digital signatures, and digital certificates.

The **Java Cryptography Architecture** provides classes and interfaces for working with public and private keys, digital certificates, message signing, digital signatures verification, accessing protected keystores, and some other processes. These classes and interfaces are located in the standard packages java.security and java.security.cert

The Most Important Classes in JCA

java.security.KeyStore—gives access to protected keystores for certificates and passwords. The keystores are represented as set of entries and each entry has a unique name, called an alias. The KeyStore class has methods for loading keystore from a stream, storing a keystore to a stream, enumerating the entries in the keystore, extracting keys, certificates and certification chains, modifying entries in the keystore, and so forth. Two major formats for storing keystores are supported—PFX (according to the PKCS#12 standard) and JKS (Java Key Store format used by JDK internally). When we create objects of the class KeyStore, the format of the keystore should be given as a parameter. The possible values are "JKS" and "PKCS12". Objects stored in a keystore can be accessed by the alias but for accessing keys a password also is required.

java.security.PublicKey—represents a public key. It holds the key itself, its encoding format, and the algorithm destined to be used with this key.

java.security.PrivateKey—represents a private key. It holds the key itself, its encoding format, and the algorithm destined to be used with this key.

java.security.cert.Certificate—it is an abstract class for all classes that represent digital certificates. It contains a public key and information for its owner. For representing each particular type of certificates (for example X.509, PGP, and so forth), an appropriate derived class is used.

java.security.cert.X509Certificate—represents an X.509 v.3 certificate. It provides methods for accessing its attributes—owner (Subject), issuer, public key of the owner, period of validity, version, serial number, digital signature algorithm, digital signature, additional extensions, and so forth. All the information in an X509Certificate object is available for reading only.

java.security.cert.CertificateFactory—provides functionality for loading certificates, certification chains, and CRL lists from a stream. The generateCertificate() method that is purposed for reading a certificate from a stream expects the certificate to be DER-encoded (according to the PKCS#7 standard) and to be in a binary or text format (Base64-encoded). For reading a certification chain, the generateCertPath() method can be used and the encoding for the chain can be specified. Acceptable are encodings such as PkiPath, that corresponds to the ASN.1 DER sequence of certificates, and PKCS7 that corresponds to the PKCS#7 SignedData object (usually, such objects are stored in files with the standard extension .P7B). It is important to take into account the fact that PKCS7 encoding does not preserve the order of the certificates and, due to this particularity, we cannot use it for storing and reading certification chains. In Java, the only standard encoding for staring certification chains is PkiPath.

java.security.GeneralSecurityException & java.security.cert.CertificateException are classes for exceptions that can be thrown when working with digital signatures and certificates.

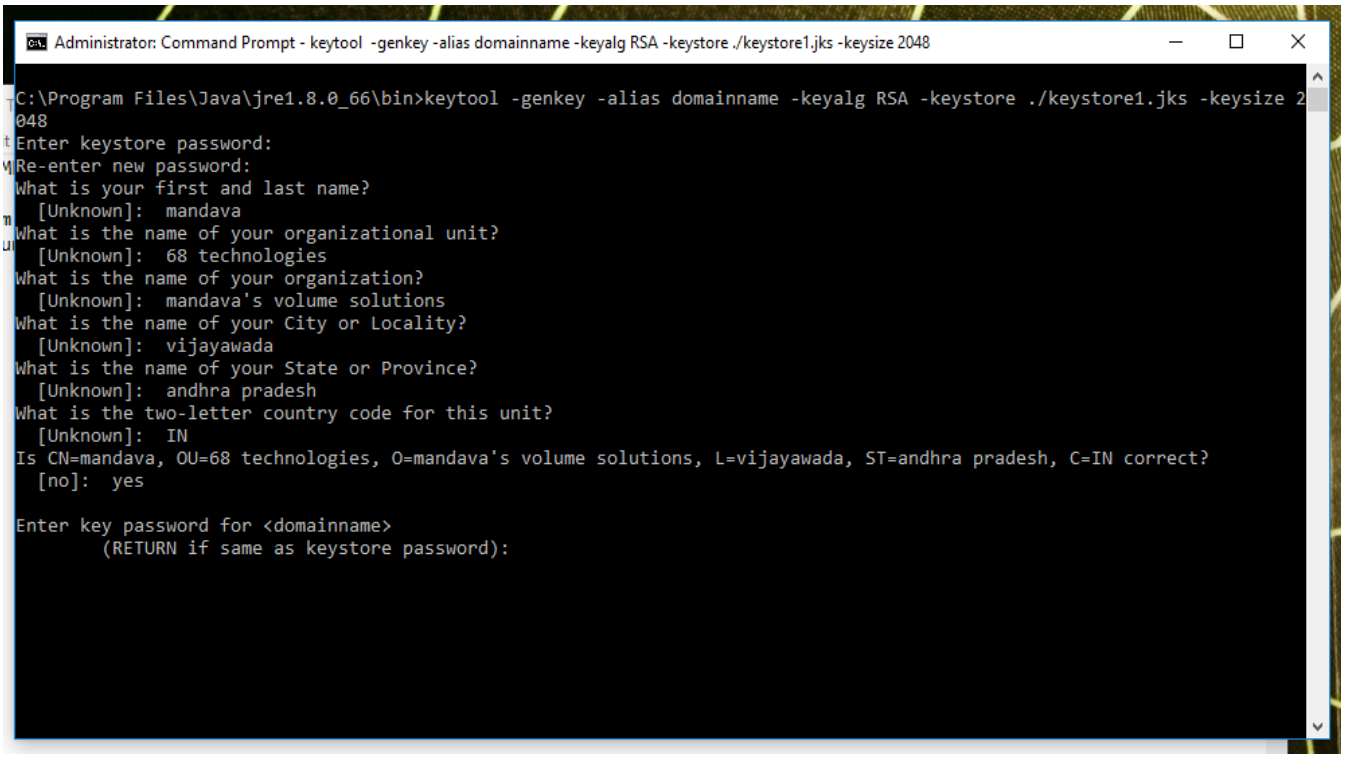
Exercise 10: Create a digital certificate of your own by using the Java Key Tool.

Requirements:

- JAVA software should be installed in your pc/laptop environment

Steps to create digital certificate:

- For windows users, the key tool path is
  C:\Program Files\Java\jre1.8.0_66\bin>
  For ubuntu/Linux users, key tool path is
  JAVA_HOME\bin\
- Now open cmd in the following path
  C:\Program Files\Java\jre1.8.0_66\bin>
  For ubuntu/Linux users, open terminal in the following path
  JAVA_HOME\bin\
- Now enter the following command
  keytool -genkey -alias domainname -keyalg RSA -keystore ./keystore1.jks -keysize 2048
- After you entering the above command you must give some information related to your certificate like below screen shot.


Administrator: Command Prompt - keytool -genkey -alias domainname -keyalg RSA -keystore ./keystore1.jks -keysize 2048

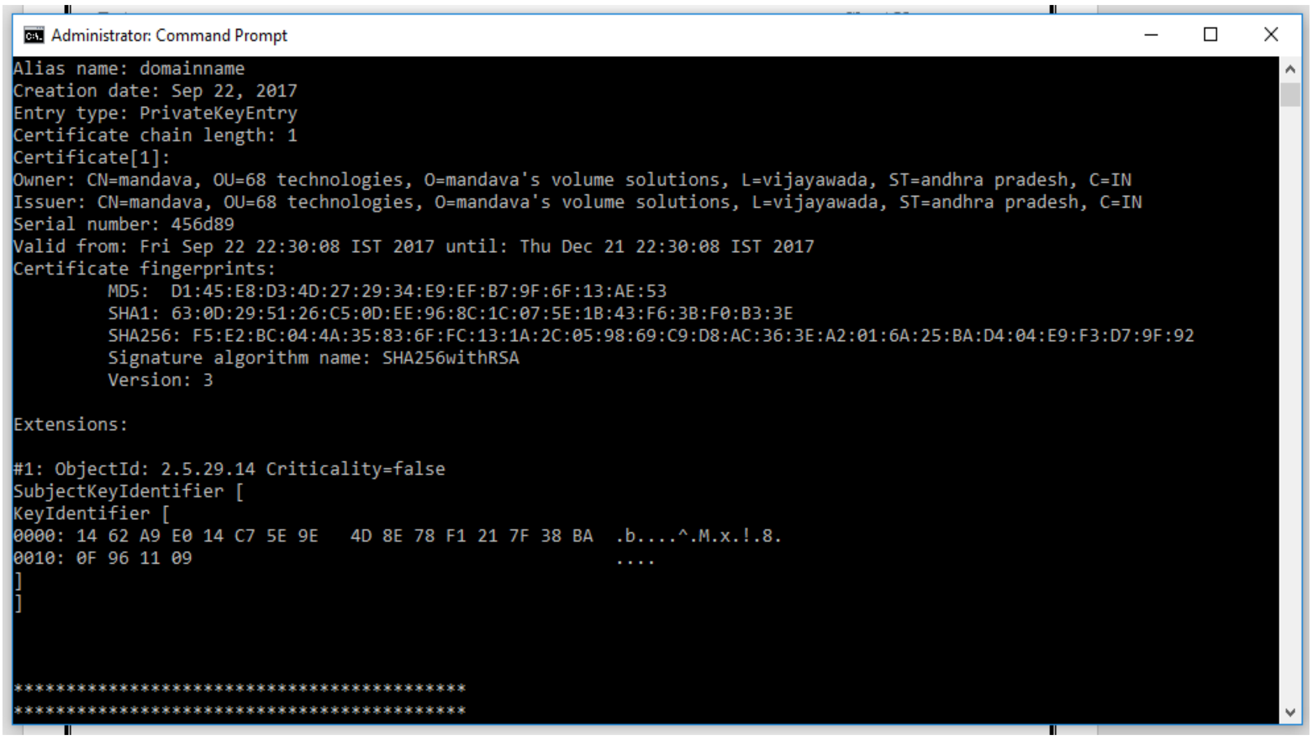
```
C:\Program Files\Java\jre1.8.0_66\bin>keytool -genkey -alias domainname -keyalg RSA -keystore ./keystore1.jks -keysize 2
048
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  mandava
What is the name of your organizational unit?
  [Unknown]:  68 technologies
What is the name of your organization?
  [Unknown]:  mandava's volume solutions
What is the name of your City or Locality?
  [Unknown]:  vijayawada
What is the name of your State or Province?
  [Unknown]:  andhra pradesh
What is the two-letter country code for this unit?
  [Unknown]:  IN
Is CN=mandava, OU=68 technologies, O=mandava's volume solutions, L=vijayawada, ST=andhra pradesh, C=IN correct?
  [no]:  yes

Enter key password for <domainname>
        (RETURN if same as keystore password):
```

- After entering all the details, you must type "YES "and enter then you must again give the password which is given above for key store.
- Now to view your created certificate you must enter the following command

C:\Program Files\Java\jre1.8.0_66\bin >keytool -list -v -keystore keystore1.jks

Output:

Now you will get the certificate as below in the screenshot with the information provided by you.



```
Alias name: domainname
Creation date: Sep 22, 2017
Entry type: PrivateKeyEntry
Certificate chain length: 1
Certificate[1]:
Owner: CN=mandava, OU=68 technologies, O=mandava's volume solutions, L=vijayawada, ST=andhra pradesh, C=IN
Issuer: CN=mandava, OU=68 technologies, O=mandava's volume solutions, L=vijayawada, ST=andhra pradesh, C=IN
Serial number: 456d89
Valid from: Fri Sep 22 22:30:08 IST 2017 until: Thu Dec 21 22:30:08 IST 2017
Certificate fingerprints:
         MD5:  D1:45:E8:D3:4D:27:29:34:E9:EF:B7:9F:6F:13:AE:53
         SHA1: 63:0D:29:51:26:C5:0D:EE:96:8C:1C:07:5E:1B:43:F6:3B:F0:B3:3E
         SHA256: F5:E2:BC:04:4A:35:83:6F:FC:13:1A:2C:05:98:69:C9:D8:AC:36:3E:A2:01:6A:25:BA:D4:04:E9:F3:D7:9F:92
         Signature algorithm name: SHA256withRSA
         Version: 3

Extensions:

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 14 62 A9 E0 14 C7 5E 9E   4D 8E 78 F1 21 7F 38 BA  .b....^.M.x.!.8.
0010: 0F 96 11 09                                        ....
]
]



*******************************************
*******************************************
```

Exercise 11: Write a Java Program to encrypt users passwords before they are stored in a database table , and to retrieve them whenever they are to be brought back for verification.

1. **PasswordUtils Java class used to encrypt and verify user password.**

```
package com.appsdeveloperblog.encryption;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.security.spec.InvalidKeySpecException;
import java.util.Arrays;
import java.util.Base64;
import java.util.Random;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;

public class PasswordUtils
{
private static final Random RANDOM = new SecureRandom();
   private static final String ALPHABET =
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
   private static final int ITERATIONS = 10000;
   private static final int KEY_LENGTH = 256;

    public static String getSalt(int length) {
      StringBuilder returnValue = new StringBuilder(length);

      for (int i = 0; i < length; i++) {

returnValue.append(ALPHABET.charAt(RANDOM.nextInt(ALPHABET.length())));
      }

      return new String(returnValue);
   }
```

```
    public static byte[] hash(char[] password, byte[] salt) {
        PBEKeySpec spec = new PBEKeySpec(password, salt, ITERATIONS,
KEY_LENGTH);
        Arrays.fill(password, Character.MIN_VALUE);
        try {
            SecretKeyFactory skf =
SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
            return skf.generateSecret(spec).getEncoded();
        } catch (NoSuchAlgorithmException | InvalidKeySpecException e) {
            throw new AssertionError("Error while hashing a password: " +
e.getMessage(), e);
        } finally {
            spec.clearPassword();
        }
    }
    public static String generateSecurePassword(String password, String salt)
 {
        String returnValue = null;
        byte[] securePassword = hash(password.toCharArray(), salt.getBytes());
        returnValue = Base64.getEncoder().encodeToString(securePassword);
         return returnValue;
    }
     public static boolean verifyUserPassword(String providedPassword,
            String securedPassword, String salt)
    {
        boolean returnValue = false;

        // Generate New secure password with the same salt
        String newSecurePassword = generateSecurePassword(providedPassword,
salt);

        // Check if two passwords are equal
        returnValue = newSecurePassword.equalsIgnoreCase(securedPassword);
```

```
         return returnValue;
    }
}
```

2. Protect User Password Code

```
package com.appsdeveloperblog.encryption;
public class ProtectUserPassword
{
            public static void main(String[] args)
            {
            String myPassword = "myPassword123";
          //Generate Salt. The generated value can be stored in DB.
        String salt = PasswordUtils.getSalt(30);
          // Protect user's password. The generated value can be stored in DB.
String mySecurePassword = PasswordUtils.generateSecurePassword(myPassword,
salt);
      // Print out protected password
            System.out.println("My secure password = " + mySecurePassword);
            System.out.println("Salt value = " + salt);
           }
}
```

Input & Output

Encrypted and Base64 encoded user password: HhaNvzTsVYwS/x/zbYXlLOE3ETMXQgllqrDaJY9PD/U=

Salt value:
EqdmPh53c9x33EygXpTpcoJvc4VXLK

Exercise 12: Write a program in java, which performs a digital signature on a given text.

Program:

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.Signature;
import sun.misc.BASE64Encoder;
public class DigSign
{
        public static void main(String[] args) throws Exception
        {
                // TODO code application logic here
                KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
                kpg.initialize(1024);
                KeyPair keyPair = kpg.genKeyPair();
                byte[] data = "Sample Text".getBytes("UTF8");
                Signature sig = Signature.getInstance("MD5WithRSA");
                sig.initSign(keyPair.getPrivate());
                sig.update(data);
                byte[] signatureBytes = sig.sign();
                System.out.println("Signature: \n" + new
                BASE64Encoder().encode(signatureBytes));
                sig.initVerify(keyPair.getPublic());
                sig.update(data);
                System.out.println(sig.verify(signatureBytes));
        }
}
```

Output:

```
Signature:
b9LojjoRIPPdnAsz0Q/8WU4DyAsMYCWmE/c9oIZPutzw2SOhDKMOY7O/Ifb1
sCRjXxPGoUmMyrjE
Sss6yRXkMh6jTs0qBALSPWxzfGQ2ZT5WiwcDi+2ipWdjQRAAGJEDixA4fZ2d
+81S3nDHVj+dvpY6
MDhoDcXm1Pt0Ne+/8TQ= true
```

Exercise 13: Study phishing in more detail. Find out which popular bank sites have been phished and how.

Phishing is the attempt to obtain sensitive information such as usernames, passwords, and credit card details (and, indirectly, money), often for malicious reasons, by disguising as a trustworthy entity in an electronic communication. The word is a neologism created as a homophone of *fishing* due to the similarity of using a bait in an attempt to catch a victim. According to the 2013 Microsoft Computing Safety Index, released in February 2014, the annual worldwide impact of phishing could be as high as US$5 billion.

Phishing is typically carried out by email spoofing[4] or instant messaging, and it often directs users to enter personal information at a fake website, the look and feel of which are identical to the legitimate one and the only difference is the URL of the website in concern. Communications purporting to be from social web sites, auction sites, banks, online payment processors or IT administrators are often used to lure victims. Phishing emails may contain links to websites that are infected with malware.

Phishing is an example of social engineering techniques used to deceive users, and exploits weaknesses in current web security. Attempts to deal with the growing number of reported phishing incidents include legislation, user training, public awareness, and technical security measures.

Phishing types:

- Spear phishing
- Clone phishing
- Whaling
- Link manipulation
- Filter evasion
- Website forgery
- Covert redirect
- Social engineering

- Phone phishing

ICICI Bank Phishing:

A few customers of ICICI Bank received an e-mail asking for their Internet login name and password to their account. The e-mail seemed so genuine that some users even clicked on the URL given in the mail to a Web page that very closely resembled the official site. The scam was finally discovered when an assistant manager of ICICI Bank's information security cell received e-mails forwarded by the bank's customers seeking to crosscheck the validity of the e-mails with the bank. Such a scam is known as 'phishing.'

GERMAN Bank Phishing:

The CYREN team detected a massive phishing attack on customers of the German bank Postbank, with more than 50,000 new phishing URLs detected within the first 24 hours. Phishing emails are traditionally sent to a massive group of people, in the hope that among the recipients are actual customers of the brand and within that group there are unsuspecting users that will click the phishing link and complete whatever information request is included. In this case the email and phishing site look very similar to the legitimate Postbank website and so it is hard for regular users to see that this is actually phishing scam and unfortunately, enough people will fall victim to the scammer’s attempts.

Phishing attacks on banks and financial institutions are very common. In fact, the number one target of phishing scams is popular payment service PayPal. In the case of this Postbank scam the unsuspecting visitor is asked to confirm his login credentials. Once the user submits his username and password on the bogus website, the scammers have obtained the user’s credentials and are able to steal his identity or sell the information to another cybercriminal.